Open Research Online

The Open University's repository of research publications
and other research outputs

ObLog: the combination of object-oriented and logic programming

Thesis

How to cite:

Watkins, John (1990). ObLog: the combination of object-oriented and logic programming. MPhil thesis The Open University.

For guidance on citations see FAQs.

© 1989 The Author

Version: Version of Record

Link(s) to article on publisher's website:
http://dx.doi.org/doi:10.21954/ou.ro.0000fc49

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online's data policy on reuse of materials please consult the policies page.

oro.open.ac.uk

OBLOG : THE COMBINATION OF OBJECT-ORIENTED AND LOGIC PROGRAMMING

John Watkins

A thesis submitted for the degree of Master of Philosophy,

October 1989

Computer Discipline

Faculty of Mathematics

The Open University

Milton Keynes MK7 6AA

U.K.

ProQuest Number: 27758434

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent on the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 27758434

Published by ProQuest LLC (2019). Copyright of the Dissertation is held by the Author.

All Rights Reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

UNRESTRICTED

OBLOG : THE COMBINATION OF OBJECT-ORIENTED AND LOGIC PROGRAMMING

John Watkins

A thesis submitted for the degree of Master of Philosophy,

October 1989

Computer Discipline

Faculty of Mathematics

The Open University

Milton Keynes MK7 6AA

U.K.

Date of submission: 15 November 1989

Date of award: 3 May 1990

DECLARATION

The work in this thesis is original unless otherwise stated.

I also declare that the work described in this thesis has not been submitted for any other degree.

ABSTRACT

Object-oriented programming has often been advocated as a means of improving and enhancing the facilities provided by a given programming environment. This thesis is concerned with an examination of the benefits of providing object-oriented facilities in the Logic programming language - Prolog. We consider these benefits from two different perspectives, specifically examining what benefits Prolog can gain from objects, and conversely, what benefits object-oriented programming can gain from Prolog.

A previously proposed model of object execution in Prolog was used as the basis of this research. In implementing this proposal we have critically examined how well the model supports the principles of object-oriented programming, and in those areas which we consider deficient, identified alternatives for improving the model which have subsequently been implemented for the purposes of assessment.

The name we have selected for our augmented system is ObLog, drawn from Ob(jects) in (Pro)log. We critically examine the suitability of ObLog in terms of object-oriented programming by implementing a series of example applications based on a Block World specification.

The thesis concludes by proposing some areas in which further research might usefully be conducted.

ACKNOWLEDGEMENTS

I wish to thank my two supervisors Mr. Mike Newton and Mr. Bennedict Heal for all their help and guidance during this research.

I would also like to thank the other members of the Computing Discipline who have provided advice and encouragement.

Finally, I am grateful for the financial support provided by the Open University, without which I would have been unable to study for this degree.

To Julie, thanks for all
the patience and support.

TABLE OF CONTENTS

Chapter 1 Introduction

1.1 Background

Object-oriented programming has often been advocated as a means of improving and enhancing the facilities provided by a given programming environment ([Rent82], [Cox84], [Stef86], for example). Established languages that have benefited from the introduction of object concepts, as demonstrated by their acceptance and use, include Simula [Birt73] - based on Algol, Objective-C [Cox84] - based on C, C++ [Stro86] - also based on C, Object Pascal [Tesl85] - based on Pascal, LOOPS [Bobr85] - based on Lisp, and KEE [Inte87], also based on Lisp.

The areas which are claimed to benefit from object-oriented programming are many and varied, and range from specification and design, through coding, and on up to maintenance of applications (see [Booc86]). Those applications which it is claimed are particularly suited to an object treatment include simulation and modelling, graphics - and more specifically WIMP (Window, Icon, Mouse, Pop up menu) systems, CAD (computer aided design), system programming and artificial intelligence research (see [Stef86], [Gull85], [Nier85], and [Banc85], for example).

The aim of this research was to investigate whether or not Logic Programming [Hogg84], as represented by Prolog [Cloc81], could benefit from the introduction of object-oriented facilities. A major goal of the work was to provide the user with object facilities while presenting them with a programming environment that appeared as similar to the original Prolog environment as possible. Another requirement of the implementation was that it should be

written in standard (Edinburgh style [Bowe82]) Prolog, without the necessity of having to alter the Prolog interpreter.

Initially, the major motivation for this work was to generate a system with which to investigate and familiarise ourselves with the object-oriented paradigm. This work was undertaken in parallel with a project conducted using a commercial object-oriented system - KEE (the Knowledge Engineering Environment [Watk86]), with aspects of the research in each area contributing to the other.

1.2 Basic Object-Oriented Principles

Considering the fact that the object-oriented paradigm is well established, there is surprisingly little agreement over exactly what features constitute an object system [Stef86]. The situation is further confused by the existence of systems that are essentially similar but subtly different from each other, such as Frames [Mins86] and Actors [Hewi73], which are discussed in section 1.3.

For the purposes of our work, we identify the basic features of an object-oriented system as follows.

1.2.1 Objects

Objects are conceptually independent entities which include the properties of process and information. In implementation terms, objects must incorporate both the specification for their procedures (often termed - **methods**), as well as the information representing their current state. An object's properties should be **encapsulated**, that is the properties should only be accessible via a defined interface, effectively insulating the end user from the implementation details of the object.

The effects of encapsulation are to minimise interdependency among objects [Snyd86], which in turn promotes the programming concept of **modularity**, allowing the independent development of objects, as well as enforcing the principles of information hiding and data abstraction [Pasc86].

1.2.2 Messages

To invoke a process associated with an object, it is necessary to communicate this requirement to the object. Such communication is expressed as sending the object a **message**. Information contained within the message specifies the required operation and any parameters required for its execution. Messages should be the only interface allowed to an object ([Rent82], for example), hence enforcing the concept of encapsulation.

1.2.3 Inheritance

Inheritance provides a mechanism by which the properties belonging to one object may be shared by other objects. For example, when a set of objects share some common **methods** or **data** a single object may be defined specifying the shared properties, which each object in the set can then inherit. Thus inheritance introduces economies of coding, and, since common information needs to be recorded only once, helps to promote consistency ([Cox84], for example).

There are many interpretations of inheritance ([Gold83], [Fike85], [Lieb86], for example), and it is possible for several different inheritance relationships between objects to be supported. KEE ([Fike85]) can be seen as an example of an object-oriented system, supporting **class**, **sub-class**, and **instance** relationships between objects. A class object can be seen as a description of one or more similar objects, containing the common properties of the objects belonging to

that class. An object representing a member of a class is termed an instance, and is related to the class object via an instance relationship (often termed an **isa** relationship). It is also possible for an object to exist that is not an instance of a class object, but which represents a specialisation of the class object. Such an entity can itself be seen as a class object, and is related to the original class via a sub-class relationship (often termed an **ako** - a kind of - relationship).

Many systems, such as Knowledge Craft [Carn87] and Actors [Lieb86] make no distinction between class and instance, simply supporting **isa** relationships between objects (see [Brac83] for a good discussion of **isa** networks).

1.3 Objects and Frames

In view of the confusion caused by the existence of similar but different systems which claim to be object like, and the nomenclature they use, I will use the next few paragraphs to state my own interpretation of the situation.

Fikes and Kehler [Fike85] use the terms object and frame interchangeably, whilst Steel [Stee86a] makes a distinction between objects and message passing systems (such as Actors - see next paragraph), and frame based systems.

Liebermann [Lieb81] claims that Actors are fundamentally objects, but exist in an environment in which there is no explicit concept of class/sub-class/instance, but only of peer objects or proxies as he terms them. This means that if an object is unable to respond to a message, it is able to redirect the call to a peer object which can respond appropriately.

My personal interpretation of these differences is as follows. An object and a frame are essentially structurally identical. That is, both allow descriptions of entities in terms of their processes and data, both support the concept of inheritance, and both support communication via messages. The only difference between the two lies in the use they are put to.

The role of frames is typically in the area of knowledge representation for knowledge based systems, where they provide a means of representing information drawn from a given domain. Inheritance in this role provides default information allowing deductive reasoning, as well as a means of explicitly indicating "real world" relationships within the knowledge base. In this role, the objects are essentially passive entities on which some active agent, such as a theorem prover or inference engine, acts to derive a result or conclusion.

The role of objects is more that of programming entities, involved in the description and implementation of processes. Inheritance in this role provides a means of providing shared default behaviour, reducing the amount of coding required, as well as allowing any changes to the code to be propagated throughout an application automatically. In this role, there is more dynamic interaction between the objects, with objects sending and receiving messages to initiate and conduct processing.

For the purposes of this thesis I consider objects and frames to be one and the same, using, as do Fikes and Kehler, the two terms interchangeably. I would even go as far as to include Actor systems in my definition of objects, since the proxy delegation mechanism effectively provides a means of sharing behaviour which is superficially similar to inheritance.

1.4 Structure of the Thesis

This thesis is structured in the following manner. Chapter 2 describes the early development of object-oriented concepts, investigates the issues involved in combining paradigms, as well as examining recent research into object-oriented programming and Prolog. Chapter 3 describes an object model proposed by Carlo Zaniolo [Zani84], which we adopted as the basis of our subsequent research. In Chapter 4, we describe the implementation of Zaniolo's proposal and examine how well the proposal supports object-oriented principles and describe our solutions to those areas of the proposal we identify as deficient. Chapter 5 describes how our augmented system can be used to implement an example application based on the Block World domain, which is used both to provide a demonstration of the systems use, and to illustrate several of the topics discussed in chapter 6. Chapter 6 contains a summary and discussion of the research, as well as presenting some areas in which future work might be conducted. Chapter 7 contains a list of the references cited in this thesis.

Chapter 2 Review Of Relevant Research

In this chapter we review research which we consider relevant to this work. We first examine the historical development of object-oriented concepts. Next, we describe examples of traditional programming languages that have been extended to include object features. Finally, we describe the research into implementing object-oriented features in Prolog.

2.1 Development of Object Concepts

Many early systems have demonstrated some characteristics that could be considered to be object-oriented. **SIMULA** ([Birt73]), an ALGOL like language, can be seen as an immediate ancestor of object-oriented programming. SIMULA was intended to be an extension of ALGOL 60, the implementation including ALGOL as a subset. From an object-oriented perspective, the important aspects of the extension include the introduction of the class, sub-class and instance concepts, as well as objects - programming entities which combined the properties of data and process.

SIMULA was conceived as a language for developing simulation applications, providing a simple means of describing real world entities and their properties in terms of objects. In this way, the language promoted an object-oriented programming style. In terms of its acceptance and use SIMULA is primarily a European language, having had few American supporters. The decline in the popularity of SIMULA is closely linked to that of ALGOL, which has been largely superceded by languages such as Pascal and Ada.

Smalltalk [Gold83] was the first language to demonstrate an explicit awareness of the concepts of object-oriented programming, including coining the term - "object oriented". Smalltalk can still be seen to be the strongest, in the sense of being the most complete and unified, example of the paradigm.

Smalltalk evolved as the software portion of the Dynabook project [Kay72], an early attempt to develop the personal computer. Smalltalk was based on an earlier language Kay had worked on - Flex, which in turn was heavily influenced by SIMULA. The **class** concept of SIMULA dominated the design, with the language becoming completely based on the idea of class as the major structural unit, with instances of classes - the objects - making up the entities found in an application. Subsequent work at the Xerox Palo Alto Research Centre led to the development of a full Smalltalk implementation, which is now widely available on many machines.

Although Smalltalk has been used to generate many commercial applications, from simulation packages to operating systems, it is not as popular or widely used as many of the traditional languages - such as Pascal. Most criticisms of Smalltalk are associated with the fact that the language is difficult to learn. This problem is generally attributed to two main causes: the size and complexity of the environment (with several thousand system objects [Kaeh86]), and the need for novice programmers to adjust to object-oriented programming concepts. Some work ([Born87], for example) indicates that people who learn Smalltalk as their first programming language find little difficulty in coming to terms with the object-oriented concepts other languages may provide.

Another criticism of Smalltalk, and of object-oriented languages in general, is that of poor execution speed (see [Watk89], for example). This is primarily due to the computational overhead associated with supporting message passing and inheritance. Although true of the earlier object-oriented languages, the more recently developed languages, such as Eiffel (described below), demonstrate very acceptable levels of performance.

Ada [Ada83] can be seen as an example of a new language which as part of its design, includes object-oriented features. Ada's generic packages enable an end user to create class like objects, specifying both data and process. The user can then subsequently define executable instances of these "templates". Ada additionally provides the concept of a package, which can be used to extend the language by generating new classes and instances, and of a task, which allows the natural expression of concurrent objects and activities.

Ada has many (primarily defence oriented) applications in commercial and military use, and due to its sponsorship by the United States Defense Department seems set to become widely used and accepted by a large programming audience.

Eiffel [Meye88] can be seen as an example of one of the most recent purely object-oriented languages. Eiffel is also a good example of how efficient (especially in terms of execution speed) an object-oriented language can be.

The main design goals for Eiffel included (as one might expect from Meyer's earlier work, see [Meye87]) software reusability and extensibility, portability, and as mentioned above, a high degree of computational efficiency. This last requirement was particularly important, since Eiffel was expected to be employed commercially to implement medium to large scale applications.

In order to achieve these aims, Eiffel is based on the principles of object-oriented design (see [Meye87], for example). Eiffel fully supports the object-oriented concepts presented in chapter 1. Facilities are provided for creating objects with the properties (or **assertions** as Meyer terms them) of state and process. Access to object properties is supported by means of messages. Class - sub-class - instance relationships are supported, as well as the mechanism of multiple inheritance.

Eiffel runs under UNIX ([UNIX84]), and is currently supported on about twenty different machine architectures. The Eiffel compiler generates C ([Rosl84]) as an intermediate language, providing both portability as well as high performance (in terms of execution speed). Further, optimisation processes within the compiler ensure that the C image of the source code is as efficient as possible - removing unnecessary code, and optimising message calls to routines.

The full Eiffel development environment provides a complete set of object-oriented tools and facilities including object-oriented browsers, tracer, and symbolic debugger. One of the most important facilities is the object library Eiffel provides which contains a large number of commonly used objects and their properties which, following Eiffel's object-oriented philosophy, developers can simply plug straight into their own applications.

2.2 Hybrid Systems

As mentioned in Chapter 1, many preexisting programming languages have been extended to include object-oriented facilities. The success of these hybrid systems can be gauged by their popularity and widespread use. Examples of such systems include Flavors [Moon86], Loops [Bobr83], ObjectLisp [Dres85], CommonLoops [Bobr86], KEE [Inte87], Objective-C [Cox84], and C++ [Stro86]. A common issue is why did these workers combine the existing language with objects in preference to simply developing a new object-oriented language from first principles.

Stefik and Bobrow [Bobr86] claim that merging an object system into an existing programming language, in their case LISP, has a number of benefits. In the first instance, the new hybrid system will have a ready made audience - the existing LISP programmers who have already mastered the language. If, as Stefik and Bobrow suggest, the hybrid system is upwardly compatible with its base language, this also allows the "incremental conversion of programs from a functional to an object-oriented style".

Another important point stressed in their paper is that of portability. If the hybrid system is based on a standard implementation of a given language, this should then also impart portability to the new system. For example, Stefik and Bobrow claim that in developing the CommonLoops system [Bobr86] in Common Lisp, and having made no alterations to the Lisp interpreter, their system is now available on most commercially available Lisp workstations.

Steels [Stee86b] examines the issue of combining multiple paradigms from a different perspective, that of knowledge representation in problem solving systems. Steels states that there is no single universal knowledge representation for solving all problems, but that different problems require different representations. Steels' solution to this problem is to combine several representation formalisms, including rule-based, logic and frame-based programming, in a single unified system - KRS (the Knowledge Representation System). The KRS system is described as providing a "glue" like mechanism which allows the combination of knowledge represented in differing formalisms to be used for the purposes of reasoning. Steels claims that KRS solves many of the problems associated with the expression of knowledge and its representation, and argues that the combination of existing formalisms is a more productive approach than attempting to develop a single all encompassing representation language.

In the case of the language C++ ([Stro86]), the motivation for introducing object-oriented extensions to the existing language - C ([Rosl84]), was based on issues of computational efficiency.

The author of the language had a requirement to write event-driven simulations which under normal circumstances Simula would have been employed to implement. In this particular instance however there existed a requirement for high execution speed, which Simula was considered unable to support.

The adopted solution was to take C, an existing high performance language (in terms of speed of execution), and introduce a set of object-oriented extensions to provide the same functionality that Simula supported in terms of object features. These features included the class concept with its property inheritance, plus the means of creating instance objects based on these classes.

Since C++ was based on C (see the appendix of [Orwe49] for one interpretation of the languages name), it retains a high degree of portability. Further advantages which this approach provide (reiterating Bobrow and Stefik's claims) include the large body of existing C library routines, and the large numbers of C programmers who would benefit from the availability of object-oriented facilities.

With the execution speed of C coupled with the facilities to generate object-oriented applications, C++ has proved to be a powerful object-oriented language. Since its origins in 1980, C++ has become increasingly popular with both existing C programmers who find the object-oriented extensions of use in extending their programming repertoire, as well as for object-oriented programmers who are unsatisfied with the execution speed of languages such as Smalltalk and Simula.

2.3 Recent Prolog Object Research

In the previous section we have seen how object-oriented features have been introduced into an existing language to correct some perceived deficiency of that language. Specifically in terms of Prolog, we identify lack of program structure [Hogg84], as well as limited knowledge representation facilities [Stee86a] as such deficiencies, and suggest that the introduction of object-oriented facilities would act to correct these deficiencies.

Several attempts have been made at incorporating object-oriented capabilities in Prolog ([Shap83b], [Zani84], [Gull85], [Anje86], [Fuka86], [Mcca86], for example). Booch [Booc86] states that some languages are better suited to the application of object-oriented concepts than others, and claims that the major issue is how well a given language is able to embody and enforce the properties of an object. The following paragraphs critically examine how well the various object models proposed by the above workers support the object paradigm. Specifically, each is assessed in terms of the following criteria : support for object features, how well the model enforces object concepts, and the role of Prolog in the implementation.

[Shap83b] Shapiro and Takeuchi demonstrate in their paper that the basic operations of object-oriented programming - including the creation of objects, message passing, class-superclass hierarchies - can be implemented in Concurrent Prolog [Shap83a]. Their system is superficially similar to Hewit's Actor systems [Hewi77], with computation performed via the cooperation of conceptually independent objects resident in the Prolog database. The suitability of the system is demonstrated by its use in simplifying the complexity of programs defining communication networks and protocols for managing shared resources.

Shapiro and Takeuchi's model of the object paradigm, especially in the definition of their inheritance system, seems heavily dependent on the features provided by Concurrent Prolog. Similarly, some aspects of their message passing mechanism are also dependent on the underlying Prolog implementation (specifically their

"incomplete message" system). In view of this dependency, it is not immediately obvious how this work can benefit the wider, more standard, Prolog community.

[Zani84] Zaniolo describes in some detail a simple, implementation independent, object-oriented Prolog system, providing clauses to support the definition and support of objects.

The aim of Zaniolo's work [Zani84] is to provide an object-oriented capability in standard Prolog (using a DEC 10 syntax [Bowe82]), whose implementation did not require alterations to be made to the Prolog interpreter. The result of this aim is the generation of a Prolog environment in which object capabilities are (optionally) available to the programmer.

Zaniolo's proposal supports the basic object-oriented features we have described in Chapter 1 in terms of providing objects, a message passing system, and a basic inheritance mechanism which supports simple "isa" relationships. These features are provided by the definition of three Prolog infix operators - **with**, **:** and **isa**, which are associated with object definition, messages and inheritance respectively (these operators are fully described in chapter 3).

[Gull85] Gullichsen describes the development of an object-oriented system which is intended to provide Smalltalk like features in a Prolog environment. The aim of the work, according to Gullichsen, was two fold: to provide a tool with which to explore the object-oriented paradigm, and to investigate how well logic programming and object-oriented programming could be integrated.

Although Gullichsen's approach seems similar to that of Zaniolo, that is in terms of the system not requiring any alteration of the interpreter and with the object features being supported by Prolog clauses, the resulting system is very different. Gullichsen's work can be seen to be an object system that has used Prolog as its implementation language, whereas Zaniolo's system can be seen as a system which is essentially a Prolog environment in which object features are provided. Consequently, Gullichsen's objects appear far removed from the underlying Prolog.

In addition, Gullichsen's model appears to violate a fundamental tenet of object-oriented programming by allowing object state to be accessed, and even altered, outside of the strict interface of message passing. Gullichsen excuses this breach of encapsulation on the grounds of "computational efficiency", presumably for applications in which the extra processing required for a message call might be considered too great.

[Anje86] Anjewierden's paper describes a series of extensions to Prolog intended to provide object like facilities, which were subsequently used to implement a programmable user interface to a computer graphics package.

Anjewierden readily admits in his paper that the system he describes does not wholly support the object paradigm, but is based more on the message passing features of objects, and their application to interfacing. This point is further demonstrated by the lack of an inheritance mechanism. The system described is

aimed more at encouraging an object-oriented style by defining a set of simple predicates whose use emulates some of the features of object-oriented programming. Encapsulation of Anjewierdens "objects" is achieved only through the level of indirection the message passing predicates provide which hides the clauses representing the object from the user.

[Mcca86] McCabe's work is essentially similar to that of Zaniolo, in that it attempts to provide object-oriented facilities in Prolog without altering the interpreter. In much the same way, McCabe provides these facilities via the definition of a set of infix operators, replacing Zaniolo's **with** operator with ' : '. McCabe also uses this operator in message calls (in much the same way Zaniolo does) as the separator between the object identifier and the specification of the required action. Inheritance is provided not as in Zaniolo's proposal, that is as individual assertions in the Prolog database, but as properties of the object, with the relationships between objects specified via arguments to the object name.

After examining the previous work in combining object-oriented programming and Prolog, we selected Zaniolo's proposal as the basis for our subsequent research. In the next chapter we outline the reasons for our choice, as well as describing the proposal in full and providing examples of its use.

Chapter 3 Zaniolo's Proposal

As mentioned in Chapter 1, our aim in this research was to investigate the benefits of combining object-oriented programming and Prolog. After examining the previous research in this area, we selected as the basis of our work a proposal for object-oriented programming in Prolog by Carlo Zaniolo [Zani84].

Our reasons for selecting this proposal were based on our requirement for a simple implementation which would require no alterations to be made to the Prolog interpreter, and that was based on Edinburgh style Prolog [Bowe82].

The aim of this Chapter is to describe Zaniolo's proposal. At this stage we make no attempt to analyse the suitability of the proposal as an object system (this is considered in chapter 4).

3.1 The Proposal

Zaniolo's proposal supports the basic object-oriented principles described in Chapter 1 in terms of providing the capability of defining objects, a message passing system, and an inheritance mechanism. These facilities are provided respectively by means of three system defined Prolog infix operators - **with**, **:** and **isa**, which we describe in the following sections.

3.1.1 Objects

Objects and their associated properties are defined in a clause by means of the operator - **with** as follows.

<object-id> with <property-list>.

In this clause **object-id** is an arbitrary Prolog predicate with zero or more arguments, and **property-list** is a Prolog list containing one or more Prolog terms in clausal form. Each of these terms is either a Prolog fact - representing object state, or a Prolog rule - representing an object method.

Before the proposed object system can use definitions of this form, the **with** clauses must be converted to an augmented internal representation within the Prolog database. In this process the facts and rules held in **property-list** are stored as separate clauses, with the object name included as an argument in the head of the clauses. The level of indirection between the **with** clauses and their internal representation provides the objects with a degree of encapsulation. To illustrate this mechanism consider the following example.

Given an object - **regular_polygon**, with properties **length_of_side** (L), **number_of_sides** (N), both facts representing the state of the object, and **perimeter** (P), a method for calculating the perimeter of **regular_polygon**, such an object might appear in its **with** clause form as follows.

```
regular_polygon (N,L) with [ (number_of_sides (N)),
                             (length_of_side (L)),
                             (perimeter (P) :- P is N * L) ].
```

In its internal representation such an object would appear as follows.

```
number_of_sides (regular_polygon (N,L), N).
length_of_side (regular_polygon (N,L), L).
perimeter (regular_polygon (N,L), P) :- P is N * L.
```

3.1.2 Messages

The requirement to communicate with an object in order to invoke a method is satisfied by means of the infix operator :, used as follows.

<object-id> : <target-property>.

Such an expression can be entered as a goal, or incorporated as a clause in a rule (including those contained within the **property-list** of a **with** clause). To illustrate the use of the message passing system, consider the following example, based on the **regular_polygon** object defined earlier. If we wish to invoke the perimeter method of **regular_polygon**, to find for example the perimeter of a regular polygon of length of side 10, and number of sides 6, we could enter the following goal.

regular_polygon (6,10) : perimeter (X).

This goal invokes the message passing code (defined by means of the : infix operator), and by a process of decomposition and recomposition generates a Prolog term of the same format as the internal representation of the target property. This term is now invoked using a "call" to the interpreter, resulting in

Prolog instantiating **X** to 60, the result calculated by the **perimeter** method. To illustrate this process, the Prolog term generated by the message passing code in the above instance would appear as follows.

perimeter (regular_polygon (6,10), X).

This term can now match against the internal representation of the specified property allowing, in this case, the **perimeter** method to be invoked (see **appendix j** for the full details of the message passing code Zaniolo presents in the proposal).

3.1.3 Inheritance

The specific inheritance capability proposed by Zaniolo supports simple "isa" relationships between objects, and is declared by means of the infix operator **isa**, with a clause of the following form.

<sub-object> isa <super-object>.

The meaning of this clause is that **sub-object** is to inherit the properties of **super-object**. These facts are held as assertions of the form - **isa (<sub-object>, <super-object>)** - in the Prolog database. As might be expected, the **isa** relationship is not symmetric. That is, inheritance of object properties is from **super-object** to **sub-object** only. The relationship is however transitive, that is, properties defined higher up in the **isa** lattice are inherited by all descendants lower down the lattice. To illustrate how **isa** is used in the inheritance system, consider the following clause.

square (L) isa regular_polygon (4,L).

The effect of this clause is to define an object called **square,** which is an example of the object **regular_polygon** which has four sides. Having now established this relationship, if we wish to find the perimeter of a square of length of side 10, we can send **square** the following message.

square (10) : perimeter (X).

The effect of this goal is for Prolog, via the message passing code, to instantiate **X** to 40 - the value calculated by the **perimeter** rule inherited from **regular_polygon.** In the same manner, we can ask square how many sides it has, that is, examine a fact representing the object's state.

square (10) : number_of_sides (X).

The effect of this goal is for Prolog to instantiate **X** to 4.

To illustrate the transitive nature of the **isa** operator consider the following example. We can define an instance of **square** in which the length of side is specified to be of length 20 by the following clause.

sq1 isa square (20).

We can now inspect the state of **sq1** by sending a message to, for example, its **number_of_sides** fact.

sq1 : number_of_sides (X).

As in the previous example, the effect of this clause is for Prolog to instantiate **X** to 4 - a fact inherited from **regular_polygon** via **square.** Similarly, we can send **sq1** a message to calculate its perimeter in the following manner.

sq1 : perimeter (X).

The value of which is 80, calculated by the method defined in **regular_polygon.**

The inheritance system Zaniolo proposes, although not explicitly stated in the text of the paper, appears to support multiple inheritance (as can be determined by examining some of the examples he provides). That is, a sub-object may inherit properties from many super-objects (providing there are no cycles generated in the lattice). Given the situation where an object can inherit the same property from more than one ancestor, it is important that some means of deciding which ancestor has highest priority exist. Zaniolo's solution is for the property to be inherited from the most recently created ancestor (as reflected by the order of the **isa** assertions in the Prolog database).

To illustrate how multiple inheritance may be utilised in the context of the inheritance system, consider the following example. Given an object - **rectangle,** with properties **length_of_base (L1), length_of_side (L2),** both of which are facts representing the state of the object, and **area (A),** a method for calculating

the area of **rectangle**, such an object might appear in its **with** clause form as follows.

```
rectangle (L1,L2) with [  (length_of_base (L1)),
                          (length_of_side (L2)),
                          (area (A) :- A is L1 * L2) ].
```

Since a square is a special case of rectangle in which all the sides are the same length, we may establish this relationship between the objects **square** and **rectangle** by the following **isa** assertion.

```
square (L)  isa  rectangle (L,L).
```

Using the object **sq1** defined earlier, we can now ask **sq1** to calculate its area by sending it the following message.

```
sq1  :  area (X).
```

The result of this goal is for Prolog to instantiate **X** to 400 - the value calculated by the **area** method defined in **rectangle**, which **sq1** has inherited via **square**. In addition to inheriting the **area** method, **sq1** also inherits the facts - **length_of_side** and **length_of_base**, whose values can be inspected by sending **sq1** a message of the following form.

```
sq1 : length_of_side (X).
        or
sq1 : length_of_base (X).
```

The effect of sending **sq1** either of these messages is for Prolog to instantiate **X** to 20.

In the next chapter we describe our implementation of this proposal, identifying the options available and the choices made in performing this task. We also examine how well the proposal and its implementation supports the principles of object-oriented programming, and describe our solutions to those areas we identify as deficient.

Chapter 4 Implementing the Proposal

As stated previously, a major goal of the implementation was to provide the user with object facilities, while presenting them with a programming environment that was as similar (in terms of facilities and appearance) to the original Prolog environment as possible. As mentioned in the introduction, to ensure portability our implementation uses only standard (Edinburgh syntax - see [Cloc81]) Prolog, and involves no alteration to the Prolog interpreter. The name we have given our system is ObLog, drawn from Ob(jects) in (Pro)log.

We have implemented Zaniolo's proposal and demonstrated that it works in several Prolog environments, including Dec10 Prolog [Bowe82], Quintus Prolog [Quin87] and Prolog2 [ESI87a]. However, the proposal specifies only certain aspects of an object system, leaving a number of issues which need to be considered in implementing a complete system. In this chapter we describe and discuss the implementation of the proposal, identify those aspects of the proposal which fail to adequately support a complete object-oriented system, and present our solutions to them.

Specifically, this chapter is structured in the following manner. Section 4.1 examines the various options available for the definition and validation of objects and their properties. Section 4.2 discusses the concept of encapsulation, and examines how well the proposal supports this important object-oriented topic. Section 4.3 discusses the absence of the concept of SELF in Zaniolo's proposal, discuses its use in object-oriented systems, and proposes one particular means of implementing SELF. Finally, section 4.4 critically discusses Zaniolo's proposal for an inheritance mechanism, examining how well, in terms of encapsulation this information is protected.

4.1 Object Definition and Validation

In this section we examine various aspects of object definition, including the options available for the creation of objects, as well as the validation of these definitions.

4.1.1 Object Definition

Our first task was to examine the options available for the definition of objects, and their subsequent conversion to the underlying representation described in chapter 3. Section 4.1.1.1 describes the facilities Zaniolo proposes in his paper for object definition. In sections 4.1.1.2 and 4.1.1.3 we describe two further options for object definition that we have identified, involving respectively an interactive dialogue and a consult-like facility, and critically discuss their suitability in view of the goals we have stated in the introduction to this chapter. All of the described options have been implemented for the purposes of assessment.

(NB: It is important to reiterate that the **with** clause object definitions that Zaniolo proposes are very different from their internal Prolog representation. As stated in chapter 3, the **with** clause definitions must first be converted to their internal representation before they can be utilised by the ObLog system. In ObLog, this function is implemented by means of an **object definition processor**, which when passed the identity of an object and its properties specified in a **with** clause, or the object identities specified in an **isa** clause, performs this process. The code for the ObLog object definition processor can be found in the appendices).

4.1.1.1 with and isa

Zaniolo's paper proposes two options for creating objects: the **with** clause, and the **isa** clause. Zaniolo proposes that **with** and **isa** are declared as Prolog operators. In this instance, the result of such a declaration is for **with** and **isa** to be defined as infix operators whose arguments are (in terms of Zaniolo's proposal), an object identity and its property list, and the identities of a sub-object and super-object respectively.

Once declared in this manner, **with** and **isa** can be subsequently defined as Prolog predicates. The practical implication of this feature, is that **with** and **isa** clauses can now be entered directly to the Prolog interpreter either interactively by the user, or by means of embedded Prolog statements within Prolog predicates (or object methods), with the result that their arguments (the object identity and property list in the case of **with**, and the sub-object and super-object identities in the case of **isa**) can be passed to the object definition processor for conversion to their internal object representation. Although Zaniolo's paper does not explicitly propose this use of **with** and **isa**, this is the pragmatic solution adopted in the early development of the ObLog system.

In terms of their actual usage within ObLog, **with** and **isa** are typically used to enter ad hoc object definitions (during the prototyping of an application, for example) directly to the interpreter. By necessity, such definitions will be relatively short, since it may be difficult to enter the definition of a large or complex object correctly using the simple editing facilities provided by the input buffer of the Prolog interpreter. As stated previously, **with** and **isa** clauses may also be used within object methods, to dynamically introduce object definitions for example.

4.1.1.2 Interactive Object Definition

Commercial object-oriented packages such as KEE [Inte87] and Frame Engine [ESI87b] typically provide some form of prompted dialogue facility for object definition. Such a system has a number of advantages, including its ease of use by naive users, the ability to check and validate object definitions as they are entered, as well as ensuring the capture of all relevant object information in a structured and logical manner.

Figure 4.1 provides one possible example of the dialogue that might be generated by such an option in ObLog, which in this instance is used to define the **regular_polygon** object described in chapter 3.

In this example system prompts are displayed in plain bold text, user input is in italic, and comments, which are ignored by the Prolog interpreter, are contained within the symbols "/* - */".

```
?-    define_object.                    /* Invoke dialogue option          */

Enter Object Name : regular_polygon(N,L).   /* Prompt for object ID          */

Enter Object State : number_of_sides (N).   /* number of sides fact          */
Enter Object State : length_of_side (L).    /* length of side fact           */
Enter Object State :                        /* User enters "return" -        */
                                            /* - for no more state           */
Enter Object Method : perimeter (P) :- P is N * L.   /* perimeter method     */
Enter Object Method :                       /* User enters "return" -        */
                                            /* - for no more methods         */

Enter Object Name :                         /* User enters "return" -        */
                                            /* - for no more objects         */

yes
?-
```

Figure 4.1 Dialogue Option Example

In terms of usage, the dialogue option is similar to that of **with**. Its main use is in creating ad hoc object definitions, whose properties, and in particular - methods, are relatively short. As mentioned in the introduction to this section, the dialogue option is of particular use to naive ObLog users.

In view of the goals stated in the introduction to this chapter, this means of object definition may not be wholly appropriate - since no such dialogue facility is available in the standard Prolog environment. However, this form of facility is of obvious use to users who may not be familiar with the ObLog system, and so might be provided as a library routine, which the user could optionally load into the system.

4.1.1.3 Consult-like Option

The process by which Prolog programs are typically created is achieved by editing clauses into a file which is subsequently loaded into the Prolog workspace using the Prolog primitive predicate - **consult**.

A consult-like facility which was capable of loading both standard Prolog clauses as well as object definitions into the ObLog system was identified as another possible option for use in creating objects. Using this mechanism, a previously edited source file containing object definitions in **with** and **isa** clause form would be read into the Prolog workspace. Object definitions would be identified and passed to the object definition processor for conversion to their internal representation, whilst standard Prolog clauses would be loaded in the normal Prolog manner.

In terms of its usage, this option is typically employed to create larger scale, preplanned ObLog programs which may contain complex objects and their properties. It is not suitable for ad hoc object definition because of the edit-consult-execute cycle it must by necessity support.

In view of the goals stated in the introduction to this chapter, this option appears to be more appropriate than the interactive dialogue option because it provides a familiar (to experienced Prolog users) means of creating and loading applications into the Prolog environment.

4.1.2 Object Validation

A further issue to consider in this section is that of the validation of user definitions of objects and their attributes. This validation encompasses referential integrity (that is, ensuring that where a message is sent to an object, that object is defined), aspects of duplication of object identity or properties in object definition, as well as checking for definitions generating cyclical inheritance problems. These aspects of object validation are discussed in the following sections.

4.1.2.1 Referential Integrity

The run time checking of referential integrity poses some problems in a Prolog based system. If a message is sent to a non-existent object, then the call will fail in a Prolog sense because it cannot be satisfied. If such an event occurs, the interpreter will backtrack and attempt to find another solution, with the result that such an error may not be detected, or may only be discovered when a given

program fails to perform as expected. It may be possible that a given ObLog program is actually relying on such behaviour to achieve its aims (see section 6.2 for example), but conversely such an event may be due to user error when writing the program - making a reference to a non existent object or property.

Rather than attempting to identify such a message call at run time, a better solution is to perform the checking during the definition process. This is in fact the solution selected in ObLog, and which is conducted in the object definition processor, where all references to objects, whether from a **consulted** file or those which have been entered interactively, are checked. References to non-existent objects are detected and an appropriate error message is displayed to the user.

Problems concerning referential integrity must also be considered when using the **isa** operator, since it is possible that one or both of the objects specified by such a clause may be undefined. Since one of the roles we have identified for **isa** is that of the creation of objects, as well as establishing stated relationships, we have selected to automatically create both the undefined object(s) as well as the specified relationship. Other options which might be implemented include only applying this solution to undefined sub-objects, or explicitly informing the user of the condition, and prompting them for information on how to proceed.

4.1.2.2 Cyclical Inheritance

Another aspect of validation of **isa** clauses involves checking for cyclical inheritance relationships. That is, where a sub-object is defined as being a

super-object of one of its super-objects. To illustrate this point, consider the following **isa** declarations.

a isa b.

b isa c.

c isa a.

The effect of ObLog processing these statements would be to introduce a cycle in to the inheritance hierarchy. Conceptually in object-oriented terms, such a situation makes no sense.

ObLog validates all **isa** statements, preventing such relationships being established, and in the above example would allow the first two statements to be entered, but would prevent the third - displaying an error message to the user.

4.1.2.3 Duplication of Object Identity

Zaniolo fails to identify the implications of attempting to enter an object definition in its **with** clause form where a definition of the object specified already exists. In such an instance, ObLog will behave as if the clause was a request to add the properties specified in the **property list** part of the **with** clause to those already defined for the object. To illustrate this point, consider the following two clauses.

mammal with [(blood_temperature (warm))].

mammal with [(body_covering (hair))].

The effect of ObLog processing these clauses would be to create the object **mammal**, possessing both the properties of **blood_temperature** and **body_covering**, as defined in the property lists of the **with** clauses. This is a

pragmatic solution to support since not all of a given objects properties may have been identified at its initial definition, and may require inclusion subsequently.

Another possible solution which has been considered (appropriate to the interactive definition options) is to warn the user about subsequent attempts to add properties to a previously defined object - allowing them the option of continuing or aborting the attempt (or in the case of a consulted file, to display such information at the end of the consult process).

The attempt to redefine an existing object property is another case which needs to be considered. For example, if we assume the above two properties for the **mammal** object have already been defined, the following property (ignoring the biological inaccuracy!) might also be entered.

mammal with [(body_covering (scales))].

One possible interpretation of such a clause might be as an instruction to assign the value **scales** to the existing property **body_covering**. Since the role of **with** is that of object definition, it seems logical to prevent its use as a means of assignment. Further, attempting to alter the value of one of an object's properties has implications in terms of the encapsulation of those properties, an area which is more fully discussed in the next section (section 4.2), where specific mechanisms for achieving this requirement are proposed.

Another interpretation might be that this clause, appearing after the previous clause specifying the value of the **body_covering** clause to be **hair**, is an error on the part of the user. In such a situation the user should be informed of the error by an

appropriate error message, and a prompt requesting how the system should proceed being displayed.

4.2 Encapsulation

One of the aims of encapsulation is to insulate the user from the necessity of being aware of the internal representation of object properties. Zaniolo's proposal, which attempts to provide encapsulation by means of the level of indirection between the **with** clause object definitions and their internal representation, provides one possible means of implementing such a mechanism.

Other internal representations achieving the same results are possible (see [Gull85] and [Maca86], for example), but either fail to satisfy our previously stated requirement of providing a familiar Prolog environment, or are essentially similar to Zaniolo's proposal. In this section we focus on attempting to improve the encapsulation of Zaniolo's proposal, and present a number of solutions by which such improvements can be achieved.

4.2.1 The Assignment Problem

As described in Chapter 3, Zaniolo proposed that the clauses representing an object, both its state and methods, should be held as assertions in the Prolog workspace. Once defined, Zaniolo does not propose any mechanism for changing an object's state. If this typical requirement is to be supported, the only means of altering an object's state is to be aware of the internal representation of the target attribute, and then to **retract** the clause representing its current state and **assert** a replacement clause to represent its altered state.

To illustrate this problem consider the following example. Given an object called **counter**, whose properties include **current_count** - an integer value (initially zero) representing object state, and **increment_count** - a method for incrementing **current_count**, such an object might appear in its **with** form as follows.

```
counter  with  [(current_count (0)),
                (increment_count (Result) :-
                                    counter : current_count (C),
                                    Result is C + 1,
                                    retract (current_count (counter,C)),
                                    assert (current_count (counter,Result)))].
```

In its internal representation the **current_count** fact appears as follows.

```
current_count (counter,0)
```

The knowledge of this structure must be employed by the user (in this case the person creating the **counter** object) within the **increment_count** method as the pattern for retracting the existing **current_count** fact, and again as the means of asserting the replacement, or incremented fact.

This need for the user to be aware of the internal representation of the objects compromises the encapsulation of the object properties, since the user may inspect or alter object state, via **assert** and **retract**, outside of the strict interface of messages.

4.2.2 Encouraging Encapsulation

A partial solution to the above problem is to create a uniform and simple means of assigning values to object state which removes the need for the user to be aware of the internal representation. This solution involves the definition of the infix operator ':=', which is used as follows.

<object> : <state> := <new-value>.

The meaning of this clause is that the current value of the property **<state>** (representing one of the "facts" contained in the objects property list) of **<object>** becomes **<new-value>**. Using this solution, our earlier **counter** object might now appear as follows.

```
counter with  [(current_count (0)),
               (increment_count (Result) :-
                                    counter : current_count (C),
                                    Result is C + 1,
                                    counter : current_count (C) := Result)].
```

With the introduction of this mechanism for assignment the user is never placed in the position of having to know any details of the underlying object representation, and so is less likely to adopt a style in which object properties are accessed outside of message calls - hence encouraging encapsulation.

Although this mechanism helps to insulate the user from the underlying representation, it is still possible for users to violate the **encapsulation** of an object by means of **assert** and **retract** if the internal representation of object properties is known.

4.2.3 Enforcing Encapsulation

To enforce the principle of encapsulation the implementation must prevent a user from inspecting or altering object properties outside message calls. A simple solution to this problem, which is particularly appropriate for use with the dialogue definition option described earlier, is to generate a "transparent" interface to the system using a meta-interpreter ([Cloc81], for example), in which it appears to the user that they are communicating directly with the Prolog interpreter, but in which any **asserts** or **retracts** dealing with object properties are prevented.

Although suitable in respect of clauses entered interactively to the interpreter, this solution fails to prevent violation of encapsulation via clauses embedded in object methods which have been **consulted** from a user edited file.

A more generally applicable means of hiding objects and their properties (in terms of all of the previously described definition options) is provided by the Prolog primitive predicates - **recorda** and **recordz** (see [Bowe82]). These predicates are similar to **assert**, but allow the storage of terms in the Prolog workspace in such a way that the terms can be inspected only by means of the **recorded** predicate, and then only if a specific **key** value is included in the call. In a similar way, the key must also be referred to if a term is to be removed from the database by the **erase** predicate. The use of these predicates also prevents the user from **listing** the object clauses, and thus effectively hides them completely.

Both of the above options (meta-interpreter and **recorda**) have been implemented and evaluated in conjunction with the assignment operator described in **4.2.1.** Since both options are suitable in respect of Zaniolo's proposal, our assessment

is based on the further requirements of encapsulation and the dynamic behaviour of the objects. The first ("transparent interface") option does not appear wholly appropriate because it fails to support complete encapsulation - object definitions can be introduced both dynamically using **assert** and **retract** within object methods via **with** clauses, as well as from within an uncontrolled file via the **consult** predicate. We have, therefore, selected the second option (that is, the **recorda** option) for further development.

4.3 SELF

In Zaniolo's proposal, a method which either examines or alters an object's state can only make reference to an object specified explicitly in the method at its definition. To illustrate this problem consider the following example.

Referring back to the **counter** object described previously, in Zaniolo's proposal we can define an instance of this object, say **counter1**, by entering the following statement.

counter1 isa counter.

The effect of this statement is for ObLog to establish a definition for the new object **counter1**, which will inherit the two properties of **current_count** and **increment_count** from **counter**. We can now enter the following message call to **counter1** to inspect the value of its **current_count** property.

counter1 : current_count (X).

The effect of this goal is for Prolog, via the message passing code, to instantiate **X** to the value held in the **current_count** property of **counter**, which is initially zero.

Similarly, if we define another instance of **counter**, say **counter2**, we could repeat the above operations, and get the same results. In the same way, we can also invoke the inherited **increment_count** method of **counter1** by entering the following goal.

counter1 : increment_count (X).

As expected, the effect of this goal is for Prolog, via the message passing code, to instantiate X to 1, the result calculated by the **increment_count** method. If we inspect the **current_count** property of **counter1** we find, also as expected, the value to be 1. If however we examine the **current_count** property of **counter2**, we find that it also has a value of 1, when in fact we would expect a value of 0.

This problem arises because both instances of **counter** inherit their **current_count** value from **counter**, which is the target object specified within its own **increment_count** method. This is a serious problem, since it severely limits the scope of inherited properties.

One solution to this problem is to allow inherited methods to be able to direct their results, when required, to the object the message was initially sent to, allowing in our **counter** example the instances (**counter1** and **counter2**) to acquire their own "local" value for the altered property that is independent of, and which replaces, the inherited value.

The solution implemented in ObLog, is to bind a special variable, named **SELF** (see [Gold83], [Lieb86], for example), to the name of the initially invoked object, which can susequently be referred to in the object's methods. This is done automatically by our augmented message passing code. In our solution, **SELF** must always be included as the last argument in the head of a method when that method is defined. Those methods that are expected to be inherited by other objects are able to refer to **SELF**, allowing the results to be directed to the initially invoked object. Consider the following example of how **counter** might appear using **SELF**.

```
counter with  [(current_count (0)),
               (increment_count (Result,SELF) :-
                                    SELF : current_count (C),
                                    Result is C + 1,
                                    SELF : current_count (C) := Result)].
```

Using the new definition of **counter**, and given the fact that **counter1** has already been defined via an **isa** assertion, we can now send the following message.

```
counter1 : increment_count (X).
```

The result of this goal is for **counter1** to acquire a local value for its **current_count** property of 1, the value calculated by the inherited **increment_count** method. If we now inspect the **current_count** values for **counter** and **counter2** both will be found to be 0.

4.4 Inheritance

Although Zaniolo's proposed inheritance system (see 3.1.3) provides a mechanism for allowing objects to inherit the properties of other objects, the proposed location of the **isa** network (specifying an object's ancestors as individually **asserted** facts in the Prolog database) is unsatisfactory because they have no protection from being inadvertently altered by means of **assert** and **retract**.

Further, since information about an object's ancestors can be considered to be a property of an object, it seems natural that the information should be held by the object in the same way as other object properties, providing the inheritance information with the same benefits of encapsulation that other object properties enjoy. An additional benefit of this solution is that, an object's ancestors can now be examined or altered via the uniform interface of message calls.

4.4.1 Ancestors

In the inheritance system we have developed, each object contains a property called **ancestors**, whose single argument is a Prolog list, in which each element is the name of one of that object's immediate ancestors (the list will be empty if the object has no ancestors). As an example, our **counter** object, which has no ancestors, might appear in its **with** clause form as follows.

```
counter with  [(ancestors ( [ ] )),
               (current_count (0)),
               (increment_count (Result,SELF) :-
                                    SELF : current_count (C),
                                    Result is C + 1,
                                    SELF : current_count (C) := Result)].
```

We can now define an instance of our **counter** object - **counter1** by entering the following object definition in its **with** form.

counter1 with [(ancestors ([counter]))].

The effect of this clause is for **counter1** to inherit all the properties of **counter**. In our implementation, as mentioned earlier, **isa** is responsible for establishing inheritance relationships between objects. With the introduction of this inheritance system, the role of **isa** is to now generate the appropriate **ancestors** fact for the specified objects. Thus the user could enter the following.

counter1 isa counter.

The effect of this clause is now entirely equivalent to the **with** clause form shown above. Should either of the objects specified in an **isa** clause not be currently defined, ObLog generates the objects and establishes the stated relationship.

For a more comprehensive example of our inheritance system see the "polygon example" in the appendices.

In terms of its characteristics, our inheritance system is entirely equivalent to that proposed by Zaniolo (as described in chapter 3). It supports multiple inheritance (the **ancestors** property is a list, and can contain many members), the priority of inheritance being governed by the order in which ancestors are added to the list (see chapter 6 for a more detailed discussion of inheritance).

Chapter 5 Block World : An Application

In this chapter we present an example of how ObLog might be used to implement a specific application. The example application selected for this exercise is the representation of a **block world** system (see [Wino85], [Wins77] and [Liet87] for example). The motivation for presenting such an application is two fold: to demonstrate to the reader the structure and function of an ObLog program, and to provide the reader with familiar examples which will be used to illustrate the topics discussed in Chapter 6. "Discussion, Conclusions and Future Work".

The selection of block world as the example application is largely based on the proven (in terms of its previous use) suitability of the domain for representation by means of an object-oriented treatment.

This chapter is structured in the following manner. Section 5.1 describes the background to the block world domain. Section 5.2 presents the formal specification of the problem which is used as the basis of the subsequent implementation. Section 5.3 briefly examines the process of object-oriented design, as applied to the block world domain. Section 5.4 presents an example of typical user interaction with the application. The chapter concludes with a summary and discussion of the implementation, examining its suitability, as well as discussing other possible implementation strategies that are available within ObLog.

5.1 Block World

The program SHRDLU ([Wino72]) was developed as a means of investigating reasoning in a simplified domain, but also encompassed aspects of robotics,

computer vision and natural language. The domain consisted of an environment populated by simple regular geometric entities such as cubes, spheres and pyramids. These entities could be arranged together in a variety of spatial relationships which were constrained by certain simple rules which were meant to reflect "real world" constraints, such as - "no block can be on top of a pyramid", for example.

SHRDLU was implemented using a frame based representation developed in Lisp, in which block world entities were represented by frames (or objects) within the program. These objects possessed attributes such as **on-top-of**, **colour**, and **size**, and could be manipulated by a **robot** entity, which was responsible for interacting with the block world. This interaction was implemented by means of methods attached to the robot object, and included processes such as placing one block on top of another block, placing a block on the table, and describing the current state of the block world.

5.2 Problem Specification

The following specification is of a simplified version of block world, and is not intended to model the functionality of the original system, but is intended to support a sufficient number of features to adequately demonstrate the use of ObLog in implementing such a system.

"A block may be either a cube, a pyramid or a cylinder. A block is either on top of the table or on top of another block. A free block is one which does not have another block on top of it. The actions permitted in this block world, and which are to be executed by means of a robot entity, are as follows.

- Create and name a new free block - which is initially placed on the table.
- Place a free block on top of the table or on top of another free block.
- Delete an existing free block.
- Describe the current state of the block world.

The following restrictions should apply to the blocks:

- A pyramid may not have another block on top of it.
- A cylinder may only be on top of another cylinder or the table.

Initially, the block world will contain just four instances of the three block types - two cubes (called c1 and c2), a pyramid (p1) and a cylinder (cyl1), all of which will be on top of the table.

As a means of demonstrating the system, any implementation based on this specification will be expected to execute the following steps.

- Describe the initial state of the block world.
- Attempt to place the cube c1 on top of the pyramid p1
  (an example of an illegal operation).
- Place the cube c1 on top of cube c2.
- Place the pyramid p1 on top of cube c1.
- Finally, describe the current state of the block world.

To execute each of these instructions an appropriate message should be sent to the robot entity. These instructions might be held in the form of an ObLog program and executed sequentially, or could be entered individually by the user via the Prolog interpreter. For illustrative purposes, the latter means of execution has been selected for this example."

5.3 Object-Oriented Design and Implementation

The process of object-oriented design is a well defined and established process (see [Booc86], [Loom87] and [Thom88], for example), and will not be reviewed here since it is beyond the scope of this thesis. Suffice it to say that we have followed such a design process, resulting in the design diagram shown in Figure 5.1.

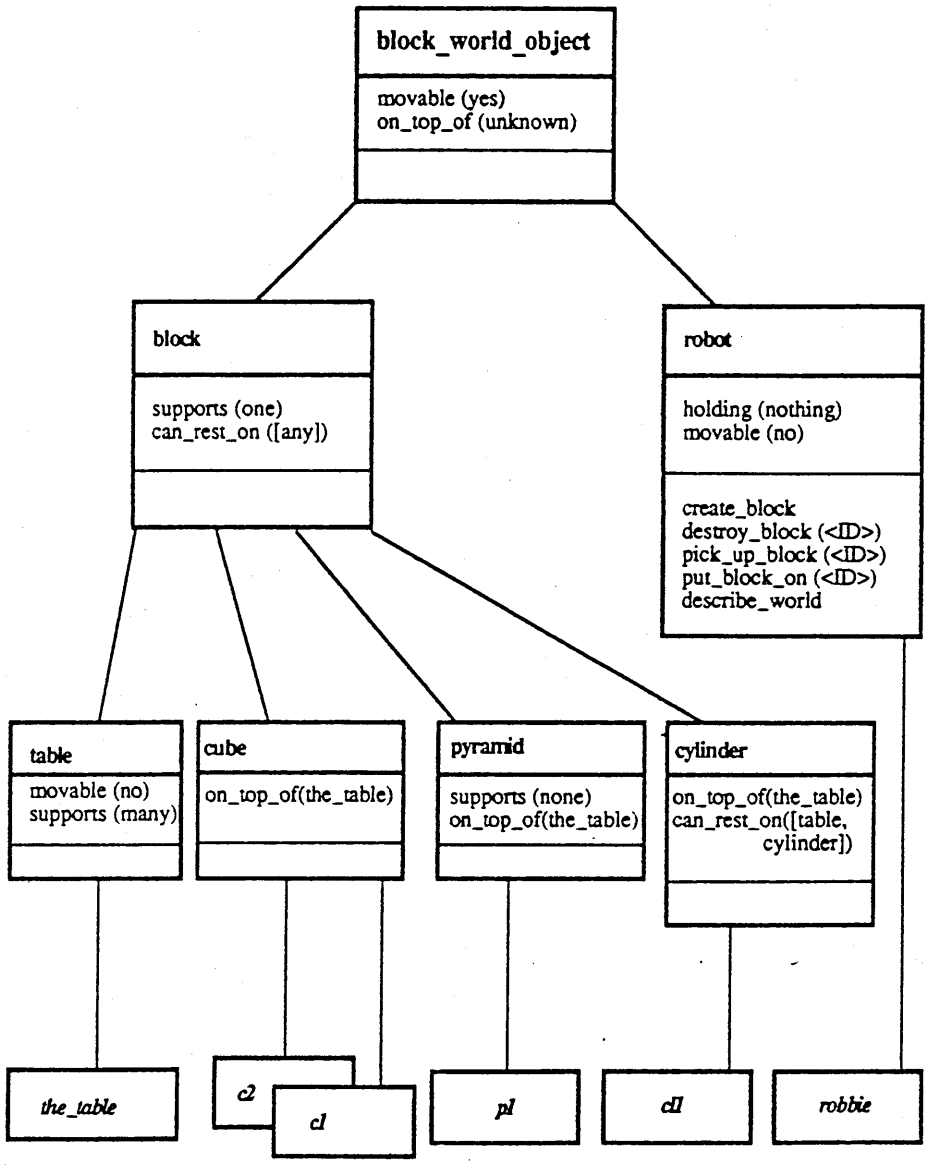


Figure 5.1 Block World Design Diagram.

The design diagram shown in figure 5.1 is based on those presented in [Loom87], and may be interpreted as follows. Objects are represented by rectangles. The top compartment of an object rectangle contains the object name. The next compartment contains the facts representing that objects state. The final compartment contains the processes or methods the object possesses. Should an object not have any state or methods, the respective compartment will be empty.

isa links are represented by the connecting lines, with the super-object above the sub-object in the figures.

These design diagrams form the basis for the implementation of the ObLog block world program presented in appendix g. Each of the entities presented in the design diagram are represented by equivalent objects within the program.

The ObLog program illustrates many of the claimed advantages of object-oriented programming. The one to one mapping between the "real world" entities and their programming counterparts appears to be a natural and easily understood one. Further, the code itself, arranged as a collection of distinct programming entities and their properties, is easy to understand - leading to code that has proven straightforward to modify and maintain.

5.4 Using the Block World System

In this section we examine how an end user might interact with the block world program. As stated earlier, we have selected an interactive dialogue as the means by which this interaction can be conducted, with the end user communicating with the robot object by entering message calls directly to the Prolog interpreter.

The example dialogue presented in figure 5.2 is based on the test tasks specified in section 5.2. In this example user input is in italic, and comments, which are ignored by the Prolog interpreter, are contained within the symbols "/* - */". All messages are directed to **robbie**, the robot object, to invoke the appropriate method.

```
?- robbie : describe_world.          /* Message to robbie to          */
   c1 is a cube, and is on top of the_table   /* invoke the describe_world  */
   c2 is a cube, and is on top of the_table   /* method                     */
   p1 is a pyramid, and is on top of the_table
   cl1 is a cylinder, and is on top of the_table
   the_table is a table, and is on top of nothing
   no

?- robbie : pick_up_block (c1).      /* Message to robbie to          */
   yes                               /* pick up block c1              */

?- robbie : put_block_on (p1).       /* An illegal operation!         */
   no

?- robbie : put_block_on (c2).       /* Message to robbie to put      */
   yes                               /* c1 on c2 - a legal operation  */

?- robbie : pick_up_block (p1).
   yes

?- robbie : put_block_on (c1).
   yes

?- robbie : describe_world.
   c1 is a cube, and is on top of c2
   c2 is a cube, and is on top of the_table
   p1 is a pyramid, and is on top of c1
   cl1 is a cylinder, and is on top of the_table
   the_table is a table, and is on top of nothing
   no
```

Figure 5.2. Sample Block World Dialogue.

The first message invokes the **describe_world** method, which is used to examine the current state of the block world, and to present this information to the user. The next user input is a message to robbie to pick up the block **c1**. **Robbie** is next requested to perform an illegal operation - placing block **c1** (a cube) on top of block **p1** (a pyramid). Because the value of the **supports** attribute of **pyramid** has the value **none**, **robbie** is unable to satisfy this goal. The next message directs **robbie** to place **c1** on top of block **c2** (another cube). **Robbie** is able to satisfy this goal. The next two messages direct **robbie** to pick up **p1** and place it on top of **c1**. Finally, **robbie** is requested to describe the current state of the block world, in which **p1** is now on top of **c1**, which is in turn on top of **c2**.

5.5 Summary and Discussion

The block world program demonstrates that ObLog is capable of supporting a frame or object based approach to a block world implementation.

Further, the implementation appears to support several of the claimed benefits of object-oriented programming. These benefits include the direct mapping between entities drawn from the application domain and their programming counterparts, and the understandability of the program code, resulting in a program that is easy to modify and maintain (see [Watk88], for example). Additionally, the program provides a high degree of flexibility, with new instances of the existing block types created using a single **isa** statement, complete with default attributes inherited from their super-objects.

This particular implementation of block world must be seen as only one of several possible means of implementing the specification.

In this implementation the objects representing the block entities perform a passive role, simply acting as a means of representing the state of the block world, while an active agent - the robot, manipulates this representation. In this sense, the implementation is similar to the sort of application that might be expected from a frame based system, such as KEE for example, in which the objects representing the blocks perform the role of a knowledge representation structure, while the separate robot object holds the methods responsible for manipulating this knowledge representation.

Another possible implementation might be based on a more purist object-oriented model, such as that provided by Smalltalk for example. In such an implementation, the objects themselves would possess both the information representing their state, as well as the methods responsible for their own manipulation. Thus to achieve a required operation, such as placing one block on another, the user might send the object to be moved a message specifying the target object on which to place itself, which might appear as follows.

c1 : put_block_on (c2).

In this instance, the meaning of the above message is that the object **c1** should place itself on top of the object **c2.**

These two options, the KEE like and Smalltalk like implementations, can be seen as extreme examples of a range of possible object-oriented implementation strategies. A further option which is open to ObLog users is that of a mixed initiative implementation, using both objects and simple Prolog predicates. This option might be attractive to naive ObLog users, allowing a degree of flexibility

not provided in a purely object-oriented environment or in a non object-oriented Prolog environment. One possible example of such a mixed initiative might involve using passive objects to represent the block world entities, while manipulation of these objects could be achieved by means of simple Prolog predicates.

Examples of both the Smalltalk and Mixed Initiative implementations, including design diagrams, example dialogue, and code, are presented in appendices h, and i respectively.

Chapter 6. Discussion, Conclusions and Future Work

As we have stated previously, one of the major motivations for this work was to investigate the benefits of combining objects and logic programming, as represented by Prolog. In this chapter we consider these benefits from two different perspectives, specifically we examine what benefits Prolog can gain from object-oriented programming, and conversely, what benefits objects can gain from Prolog.

This chapter is structured in the following manner. Section 6.1 discusses those areas in which Prolog can benefit from the introduction of object-oriented programming. Section 6.2 examines the areas in which object-oriented programming can benefit from Prolog. Section 6.3 presents a discussion of two more general object-oriented topics - encapsulation and inheritance, as related to this research. Finally, section 6.4 describes some areas in which further research might usefully be conducted.

6.1 Prolog and Objects

We identify four main areas in which Prolog can benefit from the introduction of object-oriented facilities:

- The introduction of structure in Prolog programs,
- The enhancement of Prolog's knowledge representation facilities,
- The reusability and extensibility of Prolog objects, and
- The availability of a well defined design methodology.

6.1.1 Program Structure

Lack of program structure is a well recognised problem with Prolog programs (see [Hogg84] and [Davi89], for example). This is primarily due to Prolog's reliance on the predicate as its main means of structuring programs.

Modules ([Clar84], for example) have been proposed as a means of structuring Prolog programs at a fairly high level, but are not straightforward to use, requiring extra user declarations specifying precisely how communication between modules is to take place, as well as storage as separate files which need to be loaded together to generate an application. A further important problem with modules is that there is currently no standard, with several proposals for module based systems (see also [Szer82], [Chik84] and [Okee87], for example), with widespread disagreement over precisely how they should be implemented.

Objects provide a means of factoring the Prolog database into easily identifiable programming units, whose properties are aggregated in one location. This allows programs to be more easily understood, resulting in code that is easily debugged or altered. This point is illustrated by an examination of the block world program presented in Chapter 5 (also see appendices g to i), as well as the geometric example outlined in Chapter 3 (see appendix f).

6.1.2 Knowledge Representation

Early workers in artificial intelligence believed that a single unique formalism could be found to represent all possible forms of knowledge. Since the beginning of the nineteen-eighties a consensus has developed that there is no unique representation suitable for all purposes, but that each representation has its strong and weak points (see [Stee86a], for example).

Several workers ([Liet86], [Mcde87], for example) have claimed that the knowledge representation facilities provided by logic programming are insufficient for problem solving in certain domains. Most criticisms of Prolog are based on the difficulties of representing diverse knowledge given the lack of organisation facilities provided in the Prolog database.

Frame systems ([Mins86], [Wino85]) provide a means of introducing such organisation. Such systems allow related information from the problem domain to be grouped into conceptually distinct entities - termed frames. Frames themselves can be further organised into an inheritance network, allowing information to be shared amongst related frames.

The combination of Prolog and frames represents a potentially powerful knowledge representation system. The declarative nature of Prolog combined with the structure of frames provides a representation which is capable of use in a wider range of problem domains than that covered by logic or frames individually. Additionally, Prolog's backward chaining theorem prover is available for use as an inference engine (see [Alty84], for example) for problem solving against the composite knowledge representation (see [Newt88] for a discussion of combining logic and frames for knowledge representation).

6.1.3 Reusability and Extensibility

Another area Prolog can benefit from the introduction of objects is that of the reusability of objects and their properties ([Cox84], [Booc86] and [Meye87] for example). Meyer claims that because object properties are encapsulated, and hence accessible only via a strictly defined interface, objects are an appropriate means of storing software library routines.

Meyer presents the following example. Given the fact that a programmer has a frequent requirement for storing and retrieving data from a complex data structure, it is possible for the programmer to construct an object which both contains the data structure, as well as the methods necessary to access its contents. In this way, any given program that might need to use such an object is insulated from having to be aware of exactly how the data structure is implemented, or how to access it. The object can subsequently be held in a library, and called from any application that requires such an object.

Extensibility is another area in which objects can be of benefit. If we have a requirement for an object which is similar to, but different in some respect, from our data structure example, it is a relatively simple process to create a sub-class of the object in which we can introduce the required changes. Another solution to this problem, which is appropriate for the situation in which we want to introduce some new property, is to generate a new class object containing the property, and then create an object which inherits from both the library object and the newly created one.

Meyer has demonstrated the success of this approach with the development of the object-oriented language Eiffel [Meye88], in which the concepts of reusability and extensibility are key elements of the language.

6.1.4 Object-oriented Design

The process of object-oriented design is a well defined and widely used design methodology (see [Booc86], [Loom87], [Thom88] for example). Essentially, the process relies on the direct mapping between the real world entities and their programming counterparts, resulting in the design process generating a satisfying model of reality. Booch claims that this leads to improved maintainability and

understandability for large applications.

The introduction of objects into Prolog makes the object-oriented design process, with all of its claimed benefits, optionally available to the programmer. That ObLog is capable of supporting such a methodology is demonstrated by the use of object-oriented design in the implementation of the block world example presented in chapter 5.

6.2 Objects and Prolog

In this section we examine the benefits that objects can gain from Prolog. Specifically, we identify four main areas in which objects can benefit from the features of Prolog:

- Pattern matching,
- Messages as "predicates",
- Back-tracking, and
- Messages in conjunction / disjunction.

Each of these features is discussed in the following sections. To illustrate how ObLog makes use of these features, we will use a series of examples which are based on the block world application described in chapter 5.

6.2.1 Pattern Matching in Messages

ObLog's message passing system supports the logic programming concept of **unification** ([Brat86], [Ster86], for example). Messages may be entered in which certain of the message components are uninstantiated, for which the Prolog interpreter will attempt to find a match. Specifically, of the three components of

an ObLog message call (that is, target object, target property, and arguments of the property) the target property is the only component that must be instantiated.

The fact that we can send messages in which the target object identity is uninstantiated provides a powerful extension to the object message passing concept, allowing us to issue a "broadcast message" to which any object, with the property specified in the message, can respond. For example, in our block world application, if we wish to place one block on another block, **c1** on **c2** for example, we must first establish whether **c2** already has another block on top of it. Such a requirement might be expressed as follows.

X : on_top_of (c2).

This goal can be interpreted as - is there any object in the database that has the property **on_top_of**, whose value is **c2**. The result of this goal would be for the Prolog interpreter to find the identity of the first object in the Prolog database having the required property.

As described above, we may also enter a message in which the value of the target property is uninstantiated. We have already seen instances of where just the value is uninstantiated, for example where we use a message call to inspect object state, or expect a result to be returned by a method (see **3.1.2**). A more powerful use of this feature is in combination with an uninstantiated object identifier. For example, with our previous example (any object on top of **c2**) we might now want to find "any object on top of any other object". Such a requirement might be expressed as follows.

X : on_top_of (Y).

The result of this goal would be for the Prolog interpreter, via the message passing code, to instantiate **X** to the identity of an object having the property **on_top_of**, whose value is (instantiated to) **Y**.

6.2.2 Messages as "Predicates"

If we enter a message all of whose components are instantiated, such a message can be viewed as a predicate (in the traditional computing sense, that is as a process that returns either true or false). To illustrate this point, we might wish to discover whether it was true that cube **c1** was on top of cube **c2**. Such a requirement would be expressed as follows.

c1 : on_top_of (c2).

The result of entering such a message would be either for the goal to succeed, if it were true that the **on_top_of** property of **c1** was **c2**, or to fail, if the **on_top_of** of property of **c1** was not **c2**.

This feature is of particular use in an object-oriented system because it allows the testing of such propositions in a relatively simple and easily understood manner (as compared with a procedural language based implementation, where some form of explicit test might be required).

6.2.3 Back-tracking and Messages

ObLog supports the concept of back-tracking, both in terms of Prolog clauses held within object methods, as well as in message calls. Support for the former

case is necessary in view of our aim of providing the end user with a familiar Prolog programming environment. Why ObLog should support the latter case is less obvious since in a typical object system, if for some reason a message fails to achieve its objective, no attempt is made by the system automatically to re-satisfy that goal.

In the first instance, Prolog provides this feature by utilising the interpreter's normal behaviour. However, it is still possible for a programmer to simulate the behaviour of traditional object messages by means of Prolog's "cut" mechanism, which can be used to inhibit backtracking. Zaniolo's original proposal does not support backtracking in messages because of the way he structures his message passing code. The paper gives no indication of whether or not this was a deliberate design decision.

To illustrate how we can use backtracking within message calls, consider the following example. If we want to find all objects with a given property we can enter a broadcast message (**6.2.1**), and generate all solutions by causing the goal to fail and backtrack. In our block world example we might wish to find the identity of all the blocks that are currently on top of **the_table**. Such a requirement might be expressed as follows.

X : on_top_of (the_table).

The result of this goal would be for the Prolog interpreter, via the message passing code, to instantiate **X** to the identity of an object having the required property. If we now cause the goal to fail, Prolog will attempt to generate another solution to the goal, finding other blocks (if any others exist) with the property **on_top_of (the_table)**. This is in fact the technique employed by the

describe_world method (invoked in the block world example) to examine and report the **on_top_of** facts of all the blocks that are currently defined.

6.2.4 Messages in Conjunction / Disjunction

As a consequence of ObLog supporting backtracking in message calls, it also supports conjunction in message calls. To illustrate how we might use this feature (combined with the features described above), consider the following example. If we wish to find all those objects having a combination of properties, for example blocks that are on top of **the_table**, and which have the property **supports (one)**, we might enter the following.

X : on_top_of (the_table) , X : supports (one).

The result of this conjunction would be for the Prolog interpreter, via the message passing code, to instantiate **X** to the identity of an object having both required properties. If we now cause this goal to fail, Prolog will attempt to generate another solution to the goal.

In the same way, ObLog also supports the use of disjunction in message calls. For example, we might wish to identify all those blocks that were either on top of **the_table** or were on top of another block, **c1** for example. Such a requirement might be expressed as follows.

X : on_top_of (the_table) ; X : on_top_of (c1).

The result of this disjunction would be for the Prolog interpreter, via the message passing code, to instantiate **X** to the identity of an object having either of the specified properties.

6.3 General Object Oriented Issues

In this section two more general object-oriented issues which we identify as being relevant within the scope of this work are discussed: the encapsulation of object properties, and the choice of inheritance mechanism.

6.3.1 Encapsulation

In his proposal, Zaniolo does not specifically address the problem of encapsulation. The proposal achieves some degree of information hiding by means of the message passing code which provides a level of indirection between the object structure the user perceives (the **with** declarations) and their internal representation. For reasons discussed earlier (see section 4.2), it is probable that the structure of this internal representation will become known by anyone using the system, making object properties accessible outside of the strict interface of messages. In chapter 4 we describe our solution to this problem, which ensures that object properties can only be accessed via messages.

While presenting the need for encapsulation, it may be noted that Gullichen [Gull85] appears to encourage users of his BiggerTalk system to violate encapsulation by providing a means of directly accessing instance variables outside of his message passing system. Gullichen argues that such a facility may be necessary "where computational efficiency is critical". In other words, where the overhead created by the extra message code might slow an application down too much or use up too much memory. Both are areas in which ObLog's performance can be made to suffer - depending on the application involved, and so Gullichen's approach is seductively attractive. However, I would argue that Gullichen's "computational efficiency" is an issue which in a perfect world of fast processors and plentiful memory should not affect decisions about

encapsulation. On the other hand, it could be argued that this is a question of programming style, and that the means of accessing object state directly should be provided, but that, like GOTO in procedural languages, its use should be discouraged. In ObLog's case, we have introduced an assignment operator (see section **4.2**) as a means of encouraging users not to violate encapsulation, by providing a mechanism to hide the internal representation from the user.

6.3.2 Inheritance

In this section we examine the various options that are available for implementing inheritance in object-oriented systems, and critically discuss the use of the simple **isa** inheritance scheme employed in ObLog.

In many object-oriented systems, a clear distinction exists between class - sub-class and class - instance relationships, both in the way they are represented and interpreted. The primary reason for this dichotomy is to ensure that the user is made explicitly aware of the conceptual differences between class objects and their instances, constraining the user in his use of the objects (see [Inte87], for example).

In other systems only a single inheritance mechanism is supported. The Actors system ([Hewi73], [Lieb86]) makes no distinction between class - sub-class and class - instance relationships, supporting what is essentially an **isa** link between objects. Similarly, in the knowledge representation language supported by Knowledge Craft ([Carn86]) - CRL (the Carnegie Representation Language), **isa** is again the only means of representing relationships. In both cases (Actors and CRL) it is the responsibility of the user to distinguish between objects used as classes and instances.

Like Actors and CRL, relationships between objects in ObLog are represented by **isa** links. Initially, the choice of inheritance mechanism was based on that described in Zaniolo's proposal. After examining how well **isa** was able to support the definitions of relationships between objects we were satisfied with **isa**'s performance. We claim that the distinction between class and instance objects is one which the individual user must make, essentially becoming an issue of programming style, rather than one which the system should force on the user.

6.4 Future Work

In this section we propose some further work which we consider appropriate to the research so far. This extra work is mainly aimed at enhancing the object facilities ObLog provides, specifically in terms of debugging and compiling.

6.4.1 An Object Debugger

From experience of developing applications within ObLog, it has become obvious that the existing Prolog debugger (see [Bowe82], for example) requires some enhancements to support objects. Tracing in particular (based on the "Byrd Box" model - [Byrd80]) can present problems, since not only does the debugger trace the code you are interested in, but also traces the execution of the message passing code, whose operation should ideally be transparent to the user.

Making changes to the debugger seems to conflict with one of our stated aims of portability. This problem could be avoided by generating a meta-interpreter ([Cloc81], [Eise87]) supporting an object debugger, which might be optionally loaded into the Prolog database when required.

The actual design and use of an object debugger appears to be a major task in its own right, and would involve a great deal of research into a wide range of topics from psychological issues - such as conceptual models of object execution, to more basic implementation issues - such as considering how and where it might be appropriate to implement "spy points" in an object-oriented program (See [Bray87] and [Eise88] for a good discussion of debugger requirements).

A further topic related to debugging is the provision of an "Object Browser" facility to allow the user to display and inspect the properties of specified objects. These facilities, typically provided by commercial object-oriented systems, allow the underlying object representation to be hidden from the user, displaying the object properties in a simple and easily understood manner.

6.4.2 An Object Compiler

Depending on the particular application, it is often the case that ObLog's performance, compared with an equivalent Prolog solution which does not use objects, is relatively slow and memory intensive. This problem of performance is caused primarily by the computational overhead generated by the message passing code. Since an equivalent non-object-oriented Prolog program is likely to be more efficient, in certain circumstances it would be desirable to convert an ObLog program to an equivalent non-object or non-message based Prolog representation.

The proposed system would take a source file containing the object based application and process it to generate a file containing a (non object-oriented) Prolog representation, which the user could subsequently run. It might be possible (for those Prolog implementations which support the process) to make use of the Prolog compiler ([Bowe82]) to generate a final version of the

application which would run even faster than the native Prolog. Further, it might also be possible (as in Eiffel [Meye88]) to introduce an optimisation phase within the compilation process which might for instance detect and remove unused routines or message calls (as the optimising tool provided by Eiffel does).

As with the standard Prolog compiler, compilation would only be appropriate for those applications which the user had identified as being in a final form, that is, those applications not requiring further development or testing. This is because the user will be unable to employ the debugger facilities should any problems arise in the compiled application (see [Bowe82]).

7. REFERENCES

[Ada83]
"Reference Manual for the Ada Programming Language"
ANSI/MIL-STD- 18 15A, 1983.

[Alty84]
Alty, J.L. and Coombs, M.J.,
"Expert Systems - Concepts and Examples",
National Computing Centre Limited Publications, 1984.

[Anje86]
Anjewierden, A.,
"How About a Prolog Object?",
Journee d'Etudes Languages Orientes Objets, Pages 167-176, AFCET, Paris, 1986.

[Banc85]
Bancilhon, F.,
"A Logic-Programming/Object-Oriented Cocktail",
MCC Technical Report, Number: DB-021-85, 1985.

[Birt73]
Birtwistle, G.M., Dahl, O-J, Myhrhaug, B. and Nygaard, K.,
"Simula Begin",
Van Nostrand Reinhold, New York, 1973.

[Bobr83]
Bobrow, D.G. and Stefik, M.,
"The Loops Manual"
Intelligent Systems Laboratory, Xerox Corporation, 1983.

[Bobr85]
Bobrow, D.G., et al,
"CommonLoops: Merging Common Lisp and Object-Oriented Programming"
Xerox Palo Alto Research Centre: Intelligent Systems Laboratory Series ISL-85-8, Aug. 1985.

[Bobr86]
Bobrow, D.G., et al,
"CommonLoops Merging Common Lisp and Object-Oriented Programming"
OOPSLA '86 Proceedings, Page 17-29, ACM, Sept. 1986.

[Booc86]
Booch, G.,
"Object-Oriented Development",
IEEE Transactions on Software Engineering, Pages 211-221, Vol. SE-12, Feb. 1986.

[Born87]
Borning, A. and O'Shea, T.,
"Deltatalk: an Empirically and Aesthetically Motivated Simplification of the - SmallTalk-80 Language",
Procedings of ECOOP '87, AFCET, Paris, June 1987.

[Bowe82]
Bowen, D.L., et al,
"DECsystem-10 Prolog User's Manual",
Department of Artificial Inteligence, University of Edinburgh, 1982.

[Brac83]
Brachman, R.,
"What IS-A is and isn't: -
An Analysis of Taxanomic Links in Semantic Networks",
Computer 16, Pages 30-36, 1983.

[Brat86]
Bratko, I.,
"Prolog Programming for Artificial Intelligence",
Addison-Wesley Publishing, 1986.

[Byrd80]
Byrd, L.,
"Understanding the Control Flow of Prolog Programs",
Proc. of the Logic Programming Workshop, Debrecen, Hungary, 1980.

[Carn87]
"Knowledge Craft User Manual",
Carnegie Group Inc., Station Square, Pitsburgh, USA, 1987.

[Chik84]
Chikayama, T.,
"ESP Reference Manual",
ICOT Technical Report: TR-044, 1984.

[Clar84]
Clark, K.L. and McCabe, F.G.,
"micro-PROLOG: Programming in Logic",
Prentice-Hall International, 1984.

[Cloc81]
Clocksin, W.F. and Mellish, C.S.,
"Programming in Prolog",
Springer - Verlag, 1981.

[Cox84]
Cox, Brad.J.,
"Message/Object Programming: -
An Evolutionary Change in Programming Technology",
IEEE Software, Vol.1, No.1, Pages 50-61, Jan. 1984.

[Davi89]
Davison, A.,
"Design Issues for Logic Programming-based Object-Oriented Languages",
Imperial College, Dept. of Computing Technical Report 89/9, 1989.

[Dres85]
Drescher,
"ObjectLISP User Manual"
LMI, 1000 Massachusetts Avenue, Cambridge, MA 02138, 1985.

[Eise88]
Eisenstadt, M., and Brayshaw, M.,
"The transparent Prolog Machine:
An Execution Model and Graphical Debugger for Logic Programming",
Journal of Logic Programming, Vol.5, No.4, Pages 277-342, 1988.

[ESI87a]
Expert Systems International,
"Prolog2 - Documentation",
ESI, 9 West Way, Oxford. 1987.

[ESI87b]
Expert Systems International,
"Frame Engine - Documentation",
ESI, 9 West Way, Oxford. 1987.

[Fike85]
Fikes, R. and Kehler, T.,
"The Role of Frame-Based Representation in Reasoning",
Communications of the ACM, Vol28, No.9, Sept. 1985.

[Fuka86]
Fukunaga, K. and Hirose, S.,
"An Experience with a Prolog-based Object-Oriented Language",
OOPSLA '86 Proceedings, Pages 224-231, ACM, Sept. 1986.

[Gold83]
Goldberg, A., Robson, D. and Ingalls, D.,
"SmallTalk-80: the Language and its Implementation",
Addison-Wesley, Reading, Massachusetts, 1983.

[Gull85]
Gullichsen, E.,
"BiggerTalk: Object-Oriented Prolog",
MCC Technical Report, Number: STP-125-85, Dec. 1985.

[Haye85]
Hayes, P.J.,
"The Logic of Frames"
Readings in Knowledge Representation, Ed. Brachman, R.J., and
Levesque, H.J., Morgan Kaufmann Publishers, Inc., 1985.

[Hewi73]
Hewit, C., et al,
"A Universal, Modular Actor Formalism For Artificial Inteligence",
Proc. of the 3rd International Joint Conference on Artificial Intelligence, 1973.

[Hewi77]
Hewit, C.,
"Viewing Control Structures as Patterns of Passing Messages",
Artificial Intelligence 8, 1977.

[Hogg84]
Hogger, C.J.,
"Introduction to Logic Programming",
APIC Studies in Data Processing No.21, Academic Press Inc, 1984.

[INTE87]
"KEE : The Knowledge Engineering Environment - Documentation (3.0)",
IntelliCorp Ltd., El Camino Real, Mountain View, California, USA.

[Kaeh86]
Kaehler, T. and Patterson, D.,
"A Taste Of Smalltalk"
W.W.Norton and (publishing) Company Inc., New York, 1986.

[Kay72]
Kay, A. and Goldberg, A.,
"Personal Dynamic Media",
Computer, March 1977.

[Lieb81]
Lieberman, H.,
"A Preview of Act 1",
MIT AI Laboratory AI Memo No.625, June 1981.

[Lieb86]
Liebermann, H.,
"Delegation and Inheritance: -
Two Mechanisms for Sharing Knowledge in Object-Oriented Systems",
Journee d'Etudes Languages Orientes Objets, Pages 79-89, AFCET, Paris, 1986.

[Liet86]
Lieth, P.,
"Fundemental Errors in Legal Logic Programming"
The Computer Journal, Vol.29, No.6, 1986.

[Leit87]
Leith, P.,
"A Programmed, Skeleton Formal Specification Method: The OUFDM"
The Computer Journal, Vol.30, No.4, 1987.

[Loom87]
Loomis, M.E.S., Shah, A.V. and Rumbaugh, J.E.,
"An Object Modeling Technique for Conceptual Design",
Procedings of ECOOP '87, Pages 325-335, AFCET, Paris, June 1987.

[Mcca86]
Maccabe, F.G.,
"Logic and Objects",
Imperial College, Dept. of Computing Technical Report 86/9, 1986.

[Mcde87]
McDermott, D.,
"A Critique of Pure Reason"
Computational Intelligence, 1987

[Meye87]
Meyer, B.,
"Reusability: The Case for Object-Oriented Design",
IEEE Software, Pages 50-64, March 1987.

[Meye88]
Meyer, B.,
"Object-Oriented Software Construction",
Prentice Hall, 1988.

[Mins86]
Minsky, M.,
"A Frame Work for Representing Knowledge",
The Society of Mind, New York: Simon and Schuster, 1986.

[Moon86]
Moon, D. and Keene, S.,
"New Flavours"
OOPSLA '86 Proceedings, ACM, Sept. 1986.

[Newt88]
Newton, M.A. and Watkins, J.E.,
"The Combination of Logic and Objects for Knowledge Representation"
The Journal of Object-Oriented Programming, Pages 7-10, Vol.1, No.4, 1988.

[Nier85]
Nierstrasz, O.M.,
"An Object-Oriented System",
Office Automation, Ed. Tsichritzis, D., Springer-Verlag, 1985

[Okee87]
O'Keefe, R.A.,
"Elementary Module System for DEC-10 Prolog"
Modules in Prolog, British Standards Institution, IST/5/15-Prolog, 1987.

[Orwe49]
Orwell, George,
"1984"
Secker and Warburg, London, 1949.

[Pasc86]
Pascoe, G.A.,
"Elements of Object-Oriented Programming"
BYTE Magazine, August 1986.

[Quin86]
Quintus Prolog User Manual,
Artificial Intelligence Ltd, April 1986.

[Rent82]
Rentsch, T.,
"Object-Oriented Programming",
SIGPLAN Notices, Pages 51-57, Vol.17, No.9, Sept 1982.

[Rosl84]
Rosler, L.,
"The Evolution of C - Past and Future",
AT&T Bell Laboratories Technical Journal, Vol.63, No.8 - Part 2, Oct. 1984.

[Shap83a]
Shapiro, E.Y.,
"A subset of Concurrent Prolog and its Interpreter",
ICOT Technical Report, TR-003, 1983.

[Shap83b]
Shapiro, E.Y. and Takeuchi, A.,
"Object-Oriented Programming in Concurrent Prolog",
New Generation Computing 1, Pages 25-48, 1983.

[Snyd86]
Snyder, A.,
"Encapsulation and Inhreritance in Object-Oriented Programming Languages",
OOPSLA '86 Proceedings, Pages 38-45, ACM, Sept. 1986.

[Stee84]
Steel, G.L., Jr,
"Common Lisp the Language",
Digital Press, Digital Equipument Corporation, 1984.

[Stee86a]
Steels, L.,
"AI and Programming Languages"
IFIP '86, Elsevier Science Publishers B.V (Holland), IFIP, 1986.

[Stee86b]
Steels, L.,
"Knowledge Representation System (KRS) Tutorial"
AI-Lab Tutorial, Vrije Universiteit Brussel, October, 1986.

[Stef86]
Stefik, M., and Bobrow, D.G.,
"Object-Oriented Programming: Themes and Variations",
AI Magazine, Page 40-62, Vol.6, No.4, Winter 1986.

[Ster86]
Sterling, L. and Shapiro, E.Y.,
"The Art of Prolog"
The MIT Press, Cambridge Massachusetts, 1986.

[Stro86]
Stroustrup, B.,
"The C++ Programming Language",
Addison-Wesley, 1986

[Szer82]
Szeridi, P.,
"Module Concepts for PROLOG",
SZKI Collection of Logic Programming Papers, Budapest, Hungary.

[Tesl85]
Tesler, L.,
"Object Pascal Report"
Structured Languages World, 1985.

[Thom88]
Thomas, P., Robbinson, H.M. and Emms, J.,
"Abstract Data Types : Their Specification, Representation and Use"
Oxford University Press, 1988.

[Warr77]
Warren, D.H.D.,
"Implementing Prolog - Compiling Logic Programs 1 and 2",
DAI Research Reports 39 and 40, University of Edinburgh, 1977.

[Watk86]
Watkins, J.E. and Newton, M.A.,
"KEE : The Knowledge Engineering Environment"
Open University, Computing Discipline, Technical Report 86/20, 1986

[Watk88]
Watkins, J.E. and Newton, M.A.,
"Implementing Objects in Prolog"
OOPS!, ISSN 0952-4533, BCS, March 1988.

[Watk89]
Watkins, J.E, Pitman, A.A. and Knapp, B.M.,
"Integration of Object-Oriented Programming and Knowledge Based Systems for Concurrent Simulation Applications"
MilComp '89 Conference Proceedings, Pages 17-25, Microwave Publishers Ltd., 1989.

[Wino72]
Winograd, T.,
"Understanding Natural Language",
New York : Academic Press, 1972.

[Wino85]
Winograd, T.,
"Frame Representations and the Declarative/Procedural Controversy"
Readings in Knowledge Representation, Ed. Brachman, R.J., and Levesque, H.J., Morgan Kaufmann Publishers, Inc., 1985.

[Wins77]
Winston, P.H.,
"Artificial Intelligence"
Addison-Wesley Publishing Company, 1977.

[Zani84]
Zaniolo, C.,
"Object-Oriented Programming in Prolog",
IEEE International Symposium on Logic Programming, Pages 265-270, 1984.

APPENDIX A

OBCODE.PRO - THE MESSAGE PASSING CODE

/* FILE : OBCODE.PRO **Appendix a */**

/**********

This file contains the code defining the message passing
code for the ObLog object Prolog system.
J.E.Watkins - 1986

Except for BIP's and predicates native to this file, predicates found in this file are commented with the name of the file in which that predicate may be found.

**********/

```
/* FILE : OBCODE.PRO                    Appendix a   */


/***************************************************

The message passing code, specified by the infix opperator ':'
has two clauses. The first deals with the case where the
object name is uninstantiated, and corresponds to an
"Broadcast Message", where the object ID is uninststantiated.

The second deals with the case where the object name in a
message call is instantiated. In either clause the target
property or the target value may be uninstantiated.

***************************************************/


Object : Property :-
    var(Object),
    inspect(predecessors(Object,_,_)),              /* obint */
    find_all_ancestors(Object,Ancestors),
    member(An_object,Ancestors),
    Property =.. [ID|Arguments],
    add_to_tail(Object,Arguments,New_Arguments),
    Augmented_Property =.. [ID,An_object|New_Arguments],
    testit(Augmented_Property),                      /* obint* */
    invoke(Augmented_Property).                      /* obint* */


Object : Property :-
    nonvar(Object),
    inspect(predecessors(Object,_,_)),               /* obint* */
    find_all_ancestors(Object,Ancestors),
    member(An_object,Ancestors),
    Property =.. [ID|Arguments],
    add_to_tail(Object,Arguments,New_Arguments),
    Augmented_Property =.. [ID,An_object|New_Arguments],
    testit(Augmented_Property),                      /* obint* */
    !,
    invoke(Augmented_Property).                      /* obint* */


/*****************        end   ':'      ********************/
```

```
/* FILE : OBCODE.PRO                    Appendix a   */

/************************************************

"find_all_ancestors(+any_object,-that_objects_ancestors)".

This predicate is used to generate a list of the ancestors of a
named object by inspection of the "ancestors" properties of
objects. The named object is included as the first element of
the list so generated. The predicate is non-resatifiable.

************************************************/


find_all_ancestors(Object,Ancestors)  :-
   f_a_a(Object,[],Ancestors),  !.


   f_a_a([],List,Result)  :-  reverse(List,Result).
   f_a_a([H|T],Growing_list,Result)  :-
      inspect(predecessors(H,Ancestor_list,_)),
      add_them_in(Ancestor_list,T,New_list),       /* obutil */
      add_to_head(H,Growing_list,New_Growing_list),
      f_a_a(New_list,New_Growing_list,Result).
   f_a_a(H,[],Result)  :-  f_a_a([H],[],Result).


      add_them_in([],Result,Result).
      add_them_in([H|T],Growing_list,Result)  :-
           not(member(H,Growing_list)),
           add_to_tail(H,Growing_list,New_Growing_list),
           add_them_in(T,New_Growing_list,Result).
      add_them_in([_|T],Growing_list,Result)  :-
           add_them_in(T,Growing_list,Result).

/***********      end   "find_all_ancestors"      ************/



/**************     end   file   obcode.pro          *************/
```

APPENDIX B

OBUTIL.PRO - THE OBLOG UTILITY FILE

/* FILE : OBUTIL.PRO Appendix b */

```
/**************************************************

        This file contains the utility predicates used by the
        ObLog object Prolog system, as well as
        the definition of the opertors
        with, isa, and :
        J.E.Watkins - 1986

**************************************************/
```

```
/**************************************************
```

The following operator declarations are used as "syntactic sugar" for the ObLog system.

- "with" is used in object definition.
- "isa" is used to define relationships between objects.
- ":" is used as a separator in message calls.

All three operators are define as infix operators of priority 750 (relative to with the native Prolog operators).

```
**************************************************/
```

```
?- op(750,xfy,[with,isa,:]).
```

```
/* FILE : OBUTIL.PRO                    Appendix b  */


/**************************************************

        consult_file(+object_file_name_and_extension).

This predicate is used to read into the workspace definitions
of objects held in a (previously edited) file. These objects
can be specified in their "with" clause form, or as "isa"
declarations.

**************************************************/


consult_file(File)  :-
    seeing(Input),
    see(File),
    repeat,
        read(Term),
        process(Term),
    seen,
    see(Input).



    process(Term)       :- eof(Term), !.
    process(X with Y)   :- process_object(X with Y), fail.
    process(X isa Y)    :- create_relationship(X,Y), fail.
    process(Term)       :- assert(Term), fail.



      eof(end_of_file).


/***************        end  consult_file        ***************/
```

```
/* FILE : OBUTIL.PRO                    Appendix b  */


/**************************************************

                object-definition-processor

The following predicates are used to translate the "with"
form of an object definition into its internal form, and
represents the systems "object definition processor".

**************************************************/



process_object(Object_id with Property_list) :-
        process_property_list(Object_id,Property_list),
          (inspect(predecessors(Object_id,_,_));
           store_a(predecessors(Object_id,[],_))),
        store_z(with_props(Object_id,Property_list)), !.



process_property_list(_,[]).
process_property_list(Object_id,[':-'(Head,Body)|Rest_props]) :-
     Head =.. [Funktor|Argument_list],
      New_Head =.. [Funktor,Object_id|Argument_list],
     store_a(:-(New_Head,Body)),
       process_property_list(Object_id,Rest_props).
process_property_list(Object_id,[Fact|Rest_props]) :-
     Fact =.. [Funktor|Argument_list],
        add_to_tail(_self_arg,Argument_list,New_Argument_list),
      New_Fact =.. [Funktor,Object_id|New_Argument_list],
     store_a(New_Fact),
       process_property_list(Object_id,Rest_props).



/*************     end definition processor     **********/
```

```
/* FILE : OBUTIL.PRO                    Appendix b  */

/**************************************************

                dialogue object definition option

The following code represents one possible implementation
of a dialogue definition mechanism, which allows the user to
define an object interactively from within the Prolog
environment by invoking the appropriate predicate.

"define_object" can be called with one argument - an object
definition (ie. X with Y), or with no arguments - causing a
prompted dialogue with the user to be initiated.

**************************************************/


define_object(Term)  :-
      check_with(Term), process_object(Term).
define_object  :-
    repeat,
       nl,
       write('Enter new object with property(s) or "quit" : '),
       ttyflush,
       read(Term),
       check_with(Term),
    (Term=quit;
       (process_object(Term), fail)).


check_with(Term)              :- check_with_01(Term), !.

check_with_01(X with [H|T]) :- nonvar(X).
check_with_01(quit)           :- write('Bye'), nl.
check_with_01(Term)           :-  write(Term),
                                  write(' is syntactically wrong!'),
                                  nl, fail.


/************      end   dialogue definition      ************/
```

/* **FILE : OBUTIL.PRO** **Appendix b** */

/**********

The following predicates are used to create and destroy relationships between objects. These relationships are represented by the values of the ancestors property of the objects.

"isa" can be used in several ways.

- Where both its arguments are instantiated "isa" causes the specified relationship to be established.
- Where one or both of the arguments are uninstantiated Prolog will find a match, displaying a relationship. In this case "isa" can be made to fail - generating other solutions.

create_relationship(+sub_object_id,+super_object_id).
This predicate creates the stated relationship between sub_object and super_object.

destroy_relationship(+sub_object_id,+super_object_id).
This predicate destroys the stated relationship.

**********/

```
Sub_Object isa Super_Object :-
    Sub_Object : predecessors(Ancestor_list),
    member(Super_Object,Ancestor_list).


Sub_Object isa Super_Object :-
    nonvar(Sub_Object), nonvar(Super_Object),
    create_relationship(Sub_Object,Super_Object).
```

```
/* FILE : OBUTIL.PRO                    Appendix b  */


create_relationship(Sub_Object,Super_Object)  :-
   not (Sub_Object = Super_Object),
   not (Sub_Object : predecessors(List),
        member(Super_Object,List)),
   create_object([Sub_Object,Super_Object]),
   Sub_Object : predecessors(Ancest_list),
   Sub_Object =.. [_|Args1],
   Super_Object =.. [Super_Object_id|Args2],
   process_arguments(Args1,Args2,New_Args),
   NewSupObject =.. [Super_Object_id|New_Args],
   add_to_head(NewSupObject,Ancest_list,New_Ancest_list),
   remove(predecessors(Sub_Object,Ancest_list,_)),
   store_a(predecessors(Sub_Object,New_Ancest_list,_)),  !.



    create_object([]).
    create_object([Head_Object|Rest])  :-
        Head_Object : predecessors(_),
        create_object(Rest).
    create_object([Head_Object|Rest])  :-
        Head_Object =.. [Head_Object_id|_], nl,
        write('OOPS IS CREATING THE NEW OBJECT : '),
        write(Head_Object_id),
        store_a(predecessors(Head_Object,[],_)),
        create_object(Rest).
    create_object(Object)  :-  create_object([Object]).



    process_arguments(List1,List2,Result_list)  :-
        p_a(List1,List2,[],Result_list),  !.

        p_a([],[],Result,Result).
        p_a([H|T1],[H|T2],Growing_list,Result)  :-
            add_to_tail(H,Growing_list,New_Growing_list),
            p_a(T1,T2,New_Growing_list,Result).
        p_a([],[H|T],Growing_list,Result)  :-
            add_to_tail(H,Growing_list,New_Growing_list),
            p_a([],T,New_Growing_list,Result).
```

```
/* FILE : OBUTIL.PRO                    Appendix b  */



destroy_relationship(Sub_Object,Super_Object)  :-
    remove(predecessors(Sub_Object,Ancestor_list,_)),
    excise(Super_Object,Ancestor_list,New_Ancestor_list),
    store_a(predecessors(Sub_Object,New_Ancestor_list,_)).


/******************   end   relationships   *************/
```

```
/* FILE : OBUTIL.PRO                    Appendix b  */


/************************************************
The following predicates are responsible for examining the
properties (both own and inherited) of objects.
************************************************/

display_all_properties(Object_id)  :-
   find_all_properties(Object_id,Result_list),
   show_all_properties(Object_id,Result_list).

   find_all_properties(Object_id,Result_list)  :-
      find_all_ancestors(Object_id,Ancestor_list),
      make_result(Ancestor_list,[],Result_list).

      make_result([],Result,Result).
      make_result([H|T],Growing_list,Result)  :-
         inspect(with_props(H,Prop_list)),
         add_to_tail([H,Prop_list],Growing_list,New_Growing_list),
         make_result(T,New_Growing_list,Result).

      show_all_properties(Object_id,Property_list)  :-
         nl, write('THE OBJECT '),  write(Object_id),
         write(' HAS THE FOLLOWING PROPERTIES :-'), nl, nl,
         s_a_p(Property_list).

         s_a_p([]).
         s_a_p([[Object_id,Property_list]|T])  :-
            display_an_ancestor(Object_id,Property_list),
            s_a_p(T).

            display_an_ancestor(_,[])  :-  nl.
            display_an_ancestor(Object_id,[(:-(X,_))|T])  :-
               write(X),   write('    {METHOD} FROM '),
               write(Object_id),   nl,
               display_an_ancestor(Object_id,T).
            display_an_ancestor(Object_id,[H|T])  :-
               write(H),   write('    {FACT} FROM '),
               write(Object_id),   nl,
               display_an_ancestor(Object_id,T).

/***************     end  object  examination     *************/
```

```
/* FILE : OBUTIL.PRO                    Appendix b  */

/*****************************************************

            The following predicates are system primatives.

*****************************************************/

reverse(L,L1)  :-  reverse_concatenate(L,[],L1).

   reverse_concatenate([X|L1],L2,L3)  :-
      reverse_concatenate(L1,[X|L2],L3).
   reverse_concatenate([],L,L).

member(X,[X|_]).
member(X,[_|List])  :-  member(X,List).

add_to_head(X,List,[X|List])  :-  !.

add_to_tail(X,List1,Result)  :-
   reverse(List1,List2),
   add_to_head(X,List2,List3),
   reverse(List3,Result),
   !.

append(A,B,C) :- append_01(A,B,C), !.

append_01([],R,R).
append_01([X|L1],L2,[X|L3]) :- append_01(L1,L2,L3).

excise(X,List,Result)  :-  excise_01(X,List,[],Result),  !.

excise_01(_,[],Result,Result).
excise_01(X,[X|Rest_of_list],Growing_list,Result)  :-
   excise_01(X,Rest_of_list,Growing_list,Result).
excise_01(X,[H|T],Growing_list,Result)  :-
   add_to_tail(H,Growing_list,New_list),
   excise_01(X,T,New_list,Result).


/*************  end  system  primatives  **************/
```

```
/* FILE : OBUTIL.PRO                    Appendix b  */



/*************************************************************

                    with clause object definition

The following code represents the means by which object
definitions can be entered directly by means of the with operator.
These definitions can be either entered interactively to the Prolog
interpreter, be read in from an edited file (by means of consult_file,
or can appear as clauses within object methods.

*************************************************************/


Object with Property_list :-
        check_with(Object with Property_list),
        process_object(Object with Property_list),
    !.


/***************    end    with    definition    ***************/









/***************      end of file obutil.pro      ***************/
```

APPENDIX C

OBASSN.PRO - THE OBLOG ASSIGNMENT CODE

```
/* FILE : OBASSN.PRO                    Appendix c */

/**************************************************

        This file contains the code to implement assignment
                for ObLog, the object Prolog system.
                        J.E.Watkins - 1986.

**************************************************/


/**************************************************

':='  This is the infix operator used to denote assignment.

**************************************************/


?-  op(755,xfy,[:=]).


Object : Property := New_Value :-
    Property =.. [ID,Current_Value],
    Object   =.. [Object_Name|Object_Arguments],
    uninstantiate(Object_Name,Object_Arguments,Un_Object),
    X =.. [ID,Un_Object,_,_], (remove(X); true),
    Y =.. [ID,Un_Object,New_Value,_], store_z(Y).


    uninstantiate(Obj,[],Obj).
    uninstantiate(Obj,[A],Result)      :- Result =.. [Obj,B].
    uninstantiate(Obj,[A,B],Result)    :- Result =.. [Obj,C,D].
    uninstantiate(Obj,[A,B,C],Result) :- Result =.. [Obj,D,E,F].

/******************        end  :=        ******************/


/*************     end  file  obassn.pro     **************/
```

APPENDIX D

OBINT1.PRO - RELAXED ENCAPSULATION INTERFACE

FILE : OBINT1.PRO **Appendix d */**

```
/*************************************************

This file acts to maintain a standard interface to the ObLog
object system. obint1 uses assert and retract to store and
remove object properties (obint2 uses record, recorded and
erase).

*************************************************/

store_a(Term) :- asserta(Term).

store_z(Term) :- assert(Term).

remove(Term) :- retract(Term).

inspect(Term) :- call(Term).

testit(X)      :- clause(X,_Y).

invoke(X)      :- X.

/*************     end file obint1.pro     ***************/
```

APPENDIX E

OBINT2.PRO - ENFORCED ENCAPSULATION INTERFACE

FILE : OBINT2.PRO **Appendix e */**

```
/**************************************************

This file acts to maintain a standard interface to ObLog the
Prolog object system. This interface file makes use of the
BIP's recorda, record, erase and recorded.

**************************************************/

store_a(Term) :- recorda(oops,Term,_).

store_z(Term) :- recordz(oops,Term,_).

remove(Term) :- recorded(oops,Term,Ref), erase(Ref).

inspect(Term) :- recorded(oops,Term,_).

testit(X)     :- recorded(oops,X,_);       /* either a fact, */
                 recorded(oops,(X:-Y),_).  /* or a method.   */

invoke(X)     :- (recorded(oops,X,_)); (recorded(oops,(X:-Y),_),
                 assert(X :- Y),
                 call(X),
                 retract(X :- Y));
                 ((retract(X :- Y); true), fail).


/*************     end  file  obint2.pro     **************/
```

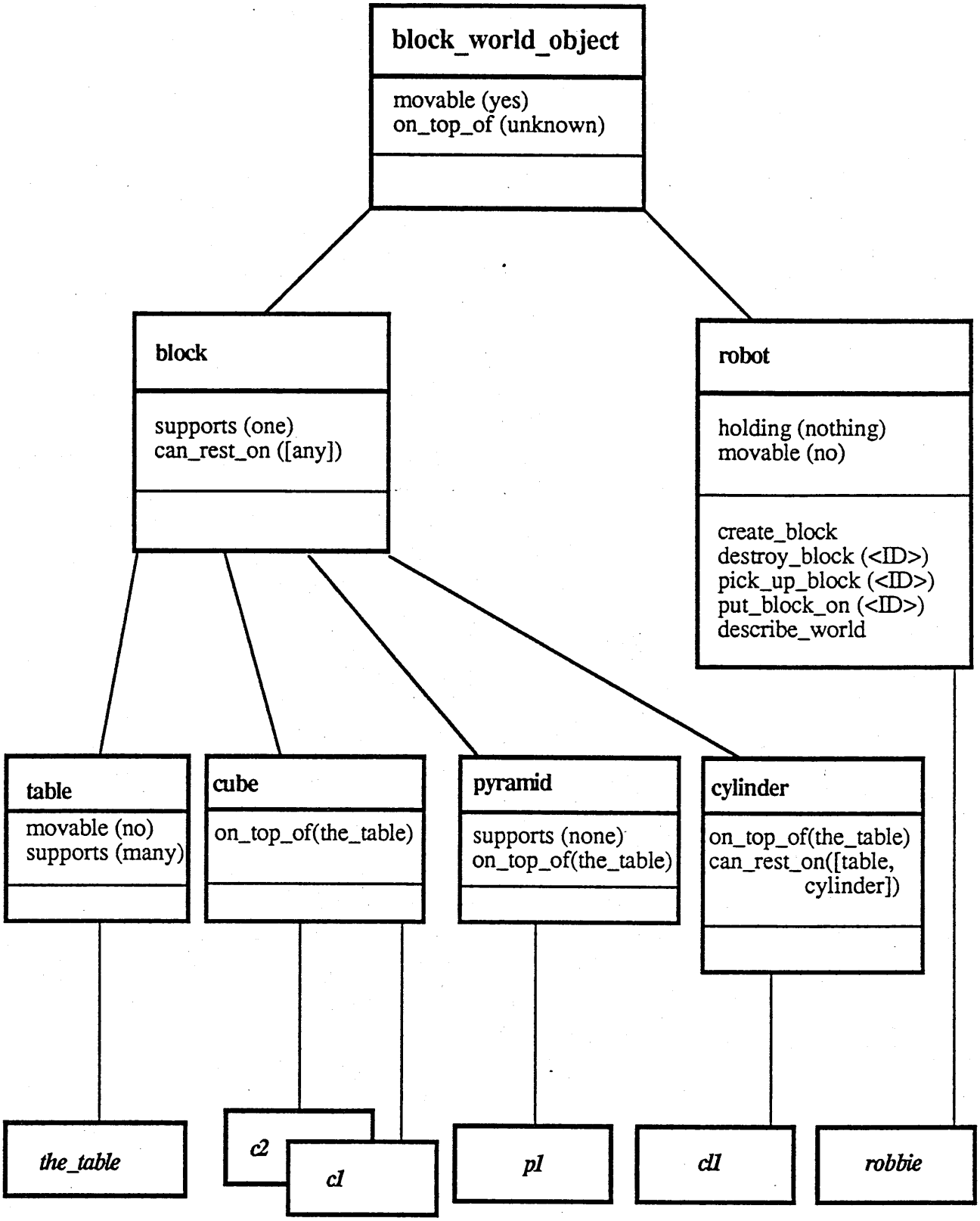
APPENDIX F

GEOMEG.PRO - THE GEOMETRIC EXAMPLE FILE

```
/* FILE : GEOMEG.PRO                    Appendix  f  */

/**************************************************

        This file  represents an implementation of the
        geometric example described in chapter 3.

**************************************************/

polygon(N)  with  [  (ancestors([])),
                     (number_of_sides(N))].


regular_polygon(N,L)  with  [ (ancestors([polygon(N)])),
                              (length_of_side(L)),
                              (perimeter(P,SELF) :- P is N * L)].


rectangle(L1,L2)  with  [ (ancestors([polygon(4)])),
                          (length_of_base(L1)),
                          (length_of_height(L2)),
                          (area(A,SELF) :- A is L1 * L2)].


square(L)   with  [ (ancestors([ regular_polygon(4,L),
                                 rectangle(L,L)]))].


/***      The following represent instance declarations      ***/


sq1   isa   square(10).          /* sq1 is a square of side 10 */
sq2   isa   square(5).

rc1   isa   rectangle(5,20).
rc2   isa   rectangle(4,25).


/*************          end file geomeg.pro          ************/
```

APPENDIX G

OPTION1.PRO - KEE LIKE IMPLEMENTATION



Design Diagram for the KEE like Implementation

```
?- robbie : describe_world.                    /* Message to robbie to          */
   c1 is a cube, and is on top of the_table    /* invoke the describe_world     */
   c2 is a cube, and is on top of the_table    /* method                        */
   p1 is a pyramid, and is on top of the_table
   cl1 is a cylinder, and is on top of the_table
   the_table is a table, and is on top of nothing
   no

?- robbie : pick_up_block (c1).                /* Message to robbie to          */
   yes                                         /* pick up block c1              */

?- robbie : put_block_on (p1).                 /* An illegal operation!         */
   no

?- robbie : put_block_on (c2).                 /* Message to robbie to put      */
   yes                                         /* c1 on c2 - a legal operation  */

?- robbie : pick_up_block (p1).
   yes

?- robbie : put_block_on (c1).
   yes

?- robbie : describe_world.
   c1 is a cube, and is on top of c2
   c2 is a cube, and is on top of the_table
   p1 is a pyramid, and is on top of c1
   cl1 is a cylinder, and is on top of the_table
   the_table is a table, and is on top of nothing
   no
```

Sample Dialogue for the KEE like Implementation

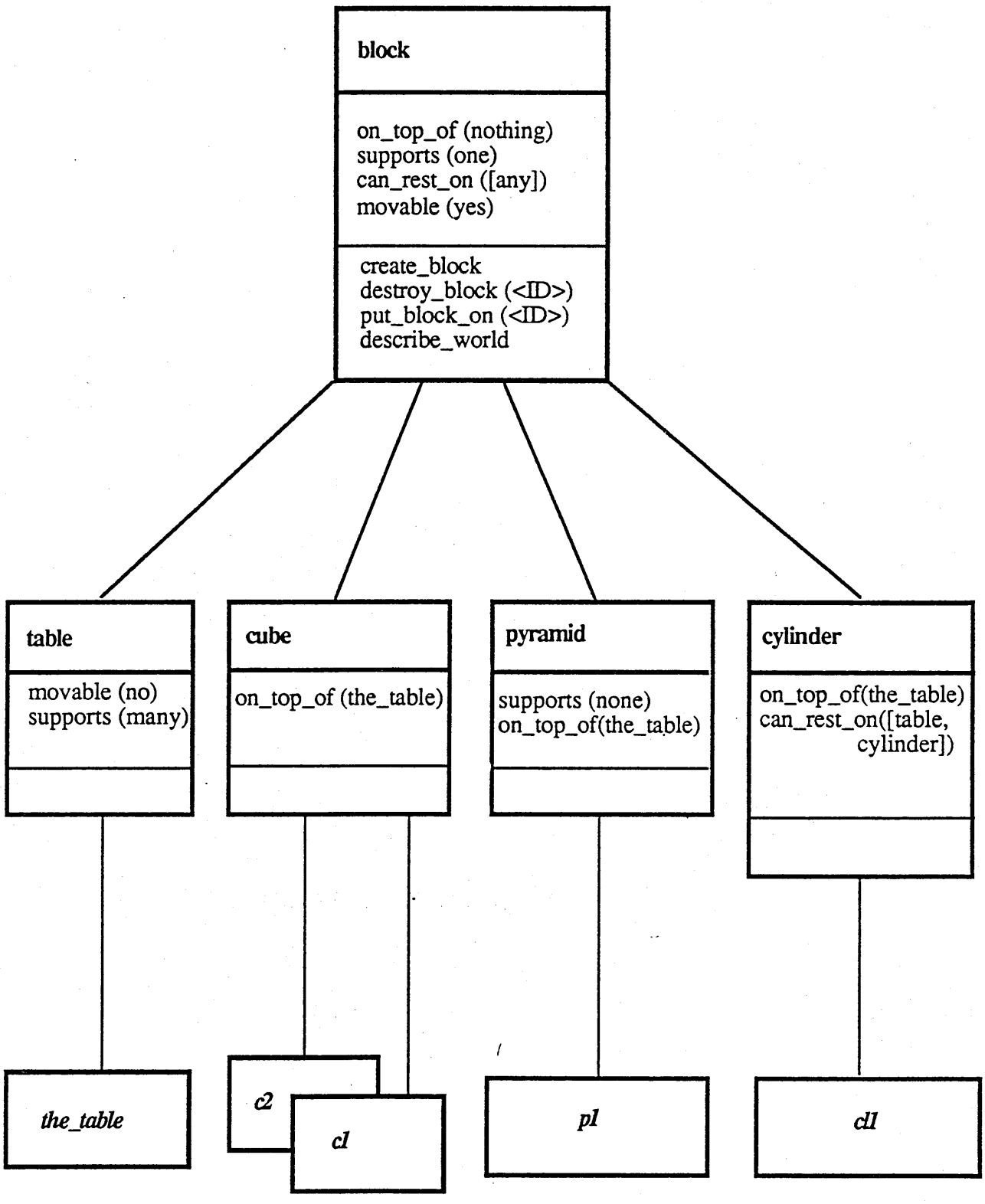
```
/* FILE : OPTION1.PRO                    Appendix g */

/**************************************************

                This file is an implementation of
                  the Kee like option of the
                  block world - chapter 5)

**************************************************/

block_world_object   with   [(movable(yes)),
                             (on_top_of(nothing))].


block with [   (ancestors([block_world_object])),
               (supports(unknown)),
               (can_rest_on([any]))].




table with [   (ancestors([block])),
               (movable(no)),
               (supports(many))].

pyramid with [    (ancestors([block])),
                  (supports(none)),
                  (on_top_of(the_table))].

cylinder with [   (ancestors([block])),
                  (can_rest_on([cylinder,table])),
                  (supports(one)),
                  (on_top_of(the_table))].

cube with [    (ancestors([block])),
               (supports(one)),
               (on_top_of(the_table))].
```

```
/* FILE : OPTION1.PRO                    Appendix g */


robot with [  (ancestors([block_world_object])),
              (holding(nothing)),
              (movable(no)),

              (pick_up_block(Target,SELF) :-
                    SELF : holding(nothing),
                    SELF : move_test(Target),
                    SELF : free_test(Target),
                    SELF : holding(_) := Block,
                    Target : on_top_of(_) := nothing,
                    Target : movable(_) := no,
                    nl, write('I Am Now Holding '),
                    write(Block), nl)),

              (put_block_on(Target,SELF) :-
                    not (SELF : holding(nothing)),
                    SELF : holding(Block),
                    SELF : supp_test(Target),
                    SELF : cpat_test(Target,Block),
                    SELF : more_test(Target),
                    Block : on_top_of(_) := Target,
                    Block : movable(_) := yes,
                    SELF : holding(Block) := nothing),

              (move_test(Target,SELF) :- !,
                    Target : movable(yes)),

              (free_test(Target,SELF) :- !,
                    not (X : on_top_of(Target))),

              (supp_test(Target,SELF) :- !,
                    not (Target : supports(none))),

              (cpat_test(Target,Block,SELF) :- !,
                    Block : can_rest_on(List),
                    Target isa Block_type,
                    (member(any,List);
                      member(Block_type,List))),
```

```
/* FILE : OPTION1.PRO                    Appendix g */


        (more_test(Target,SELF) :-
              (Target : supports(many));
              (Target : supports(one),
                not (X : on_top_of(Target))));
              (!, fail)),

        (describe_world(SELF) :-
              SELF : holding(Block),
              nl, write('I Am Holding '), write(Block),
              !,
              X isa block, Y isa X, Y : on_top_of(Z),
              nl, write(Y), write(' Is A '), write(X),
              write(', And Is On Top Of '), write(Z),
              fail)].
                                    /* end object robot */


/***    The following represent instance declarations    ***/


robbie  isa  robot.

the_table  isa  table.

c1  isa  cube.
c2  isa  cube.

p1  isa  pyramid.

cl1  isa  cylinder.



/***********          end file option1.pro          ***********/
```

APPENDIX H

OPTION2.PRO - SMALLTALK LIKE IMPLEMENTATION



Design Diagram for the SmallTalk like Implementation

```
?- the_table : describe_world.                    /* Message to the_table to   */
   c1 is a cube, and is on top of the_table       /* invoke the describe_world */
   c2 is a cube, and is on top of the_table       /* method!                   */
   p1 is a pyramid, and is on top of the_table
   cl1 is a cylinder, and is on top of the_table
   no

?- c1 : put_block_on (p1).                        /* Message to c1 to put itself */
   no                                             /* on p1- an illegal move!     */

?- c1 : put_block_on (c2).                        /* Message to c1 to put itself */
   yes                                            /* on c2 - a legal move!       */

?- p1 : put_block_on (c1).                        /* Message to p1 to put itself */
   yes                                            /* on c1 - which is on c2      */

?- p1 : describe_world.
   c1 is a cube, and is on top of c2
   c2 is a cube, and is on top of the_table
   p1 is a pyramid, and is on top of c1
   cl1 is a cylinder, and is on top of the_table
   the_table is a table, and is on top of nothing
   no
```

Sample dialogue for the SmallTalk like Implementation

```
/* FILE : OPTION2.PRO                    Appendix h */

/**************************************************

            This file is an implementation of
             the SmallTalk like option of the
                block world - chapter 5.

**************************************************/

block with [(movable(yes)),              /* These clauses */
            (on_top_of(nothing)),        /*     represent */
            (supports(one)),             /*  object state */
            (can_rest_on([any])),

            (put_block_on(Target,SELF) :-
                    SELF : move_test(SELF),
                    SELF : supp_test(Target),
                    SELF : free_test(SELF),
                    SELF : cpat_test(Target),
                    SELF : more_test(Target),
                    SELF : on_top_of(_) := Target),


            (move_test(Target,SELF) :- !,
                    Target : movable(yes)),

            (supp_test(Target,SELF) :- !,
                    not (Target : supports(none))),

            (free_test(Target,SELF) :- !,
                    not (X : on_top_of(Target))),

            (cpat_test(Target,SELF) :- !,
                    SELF : can_rest_on(List),
                    Target isa Block_type,
                    (member(any,List);
                      member(Block_type,List))),
```

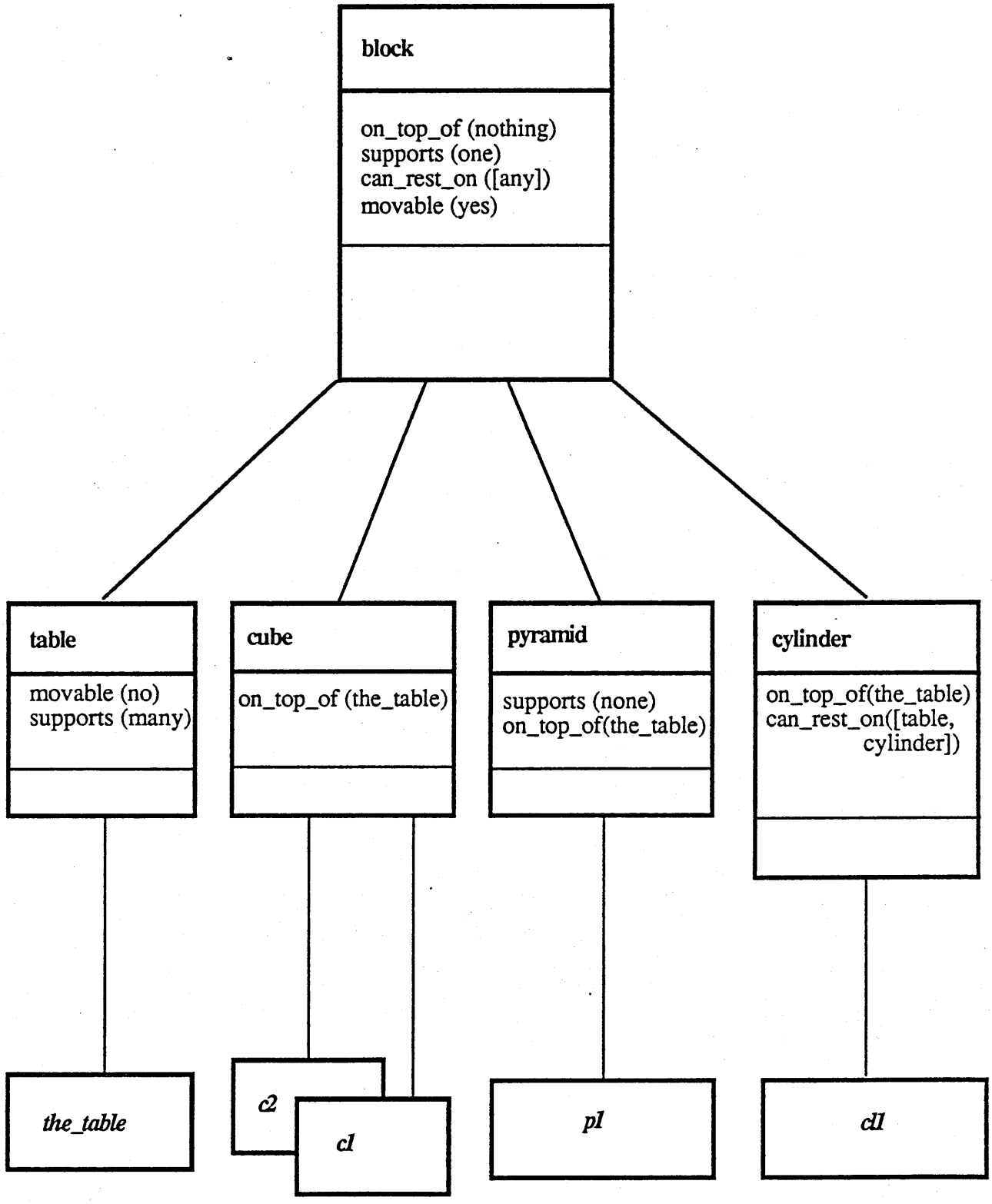
/* FILE : OPTION2.PRO **Appendix h */**

```
(more_test(Target,SELF) :- !,
        (Target : supports(many));
        (Target : supports(one),
          not (X : on_top_of(Target))));
        (!, fail)),

(describe_world(SELF) :-
        X isa block, Y isa X, Y : on_top_of(Z),
        nl, write(Y), write(' is a '), write(X),
        write(', and is on top of '),  write(Z),
        fail)].                    /* end block */
```

```
/* FILE : OPTION2.PRO                    Appendix h */


/***   The following represent sub-classes of block   ***/

table with [(ancestors([block])),
            (movable(no)),
            (supports(many))].

pyramid with [ (ancestors([block])),
               (supports(none)),
               (on_top_of(the_table))].

cylinder with [ (ancestors([block])),
                (can_rest_on([cylinder,table])),
                (on_top_of(the_table))].

cube with [ (ancestors([block])),
            (on_top_of(the_table))].


/***   The following represent instance declarations   ***/

the_table  isa  table.

c1  isa  cube.
c2  isa  cube.

p1  isa  pyramid.

cl1  isa  cylinder.


/***************  end  file  option2.pro  ***************/
```

APPENDIX I

OPTION3.PRO - MIXED INITIATIVE IMPLEMENTATION



Design Diagram for the Mixed Initiative Implementation

```
?- describe_world.                          /* Goal to invoke the          */
   c1 is a cube, and is on top of the_table  /* describe_world predicate    */
   c2 is a cube, and is on top of the_table
   p1 is a pyramid, and is on top of the_table
   c1 is a cylinder, and is on top of the_table
   the_table is a table, and is on top of nothing
   no

?- put_block _on(c1,p1).                    /* Goal to put block c1 on -   */
   no                                       /* block p1 - Illegal!         */

?- put_block_on (c1,c2).                    /* Goal to put  block c1 on -  */
   yes                                      /* block c2                    */

?- put_block_on (p1,c1).                    /* Goal to put block p1 -      */
   yes                                      /* on  block c2                */

?- describe_world.
   c1 is a cube, and is on top of c2
   c2 is a cube, and is on top of the_table
   p1 is a pyramid, and is on top of c1
   c1 is a cylinder, and is on top of the_table
   the_table is a table, and is on top of nothing
   no
```

Sample Dialogue for the Mixed Initiative Implementation

```
/* FILE : OPTION3.PRO                    Appendix i */

/**************************************************

            This file is an implementation of the
              mixed initiative option of the
                block world - chapter 5.

**************************************************/

block with [(movable(yes)),              /* These clauses */
            (on_top_of(nothing)),        /*     represent */
            (supports(one)),             /*  object state */
            (can_rest_on([any]))].


/***  The following represent sub-classes of block  ***/

table with [(ancestors([block])),
            (movable(no)),
            (supports(many))].

pyramid with [(ancestors([block])),
              (supports(none)),
              (on_top_of(the_table))].

cylinder with [(ancestors([block])),
               (can_rest_on([cylinder,table])),
               (on_top_of(the_table))].

cube with [ (ancestors([block])),
            (on_top_of(the_table))].

/***  The following represent instance declarations  ***/

the_table  isa  table.

c1  isa  cube.
c2  isa  cube.

p1  isa  pyramid.

cl1  isa  cylinder.
```

```
/* FILE : OPTION3.PRO                    Appendix i */

/**************************************************

The following represents the "external agent" which acts to
inspect or alter the knowledge representation defined above.

     **************************************************/


put_block_on(Block,Target) :-
            move_test(Block),
            supp_test(Target),
            free_test(Block),
            cpat_test(Block,Target),
            more_test(Target),
            Block : on_top_of(_) := Target.

move_test(Block) :- !, Block : movable(yes).

supp_test(Target) :- !, not(Target : supports(none)).

free_test(Block) :- !, not (X : on_top_of(Block)).

cpat_test(Block,Target) :- !,
            Block : can_rest_on(List),
            Target isa Block_type,
            (member(any,List); member(Block_type,List)).

more_test(Target) :- !,
            Target : supports(many);
            (Target : supports(one),
             not (X : on_top_of(Target)));
            (!, fail).


describe_world :-
            X isa block, Y isa X, Y : on_top_of(Z),
            nl, write(Y), write(' is a '), write(X),
            write(', and is on top of '), write(Z), fail.


/**************  end  file  option3.pro  **************/
```

APPENDIX J

ZCODE.PRO - ZANIOLO'S MESSAGE PASSING CODE

```
/* FILE : ZCODE.PRO                    Appendix j   */


/**************************************************

This file contains the message passing code for Carlo
Zaniolo's proposed Prolog object system. This code is not
used in ObLog, but is replaced by message code which
supports backtracking in message calls.

**************************************************/



Object : Property :-
    inspect(predecessors(Object,_,_)),
    find_all_ancestors(Object,Ancestors),
    member(An_object,Ancestors),
    Property =.. [ID|Arguments],
    add_to_tail(Object,Arguments,New_Arguments),
    Augmented_Property =.. [ID,An_object|New_Arguments],
    testit(Augmented_Property),
    invoke(Augmented_Property).



/******************        end  ':'     *********************/

/**************    end  file  zcode.pro        *************/
```